THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Hardware-Aware Algorithm Designs for Efficient Parallel and Distributed Processing

CHARALAMPOS STYLIANOPOULOS

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2020

Hardware-Aware Algorithm Designs for Efficient Parallel and Distributed Processing

CHARALAMPOS STYLIANOPOULOS

Copyright © 2020 Charalampos Stylianopoulos
except where otherwise stated.
All rights reserved.

ISBN 978-91-7905-360-4
Doktorsavhandlingar vid Chalmers tekniska högskola, Ny serie nr 4827.
ISSN 0346-718X

Technical report 190D
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg, Sweden
Phone: +46 (0)31-772 10 00

Author e-mail: `chasty@chalmers.se`

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2020.

Hardware-Aware Algorithm Designs for Efficient Parallel and Distributed Processing

Charalampos Stylianopoulos
Department of Computer Science and Engineering, Chalmers University of Technology

ABSTRACT

The introduction and widespread adoption of the Internet of Things, together with emerging new industrial applications, bring new requirements in data processing. Specifically, the need for timely processing of data that arrives at high rates creates a challenge for the traditional cloud computing paradigm, where data collected at various sources is sent to the cloud for processing. As an approach to this challenge, processing algorithms and infrastructure are distributed from the cloud to multiple tiers of computing, closer to the sources of data. This creates a wide range of devices for algorithms to be deployed on and software designs to adapt to.

In this thesis, we investigate how hardware-aware algorithm designs on a variety of platforms lead to algorithm implementations that efficiently utilize the underlying resources. We design, implement and evaluate new techniques for representative applications that involve the whole spectrum of devices, from resource-constrained sensors in the field, to highly parallel servers. At each tier of processing capability, we identify key architectural features that are relevant for applications and propose designs that make use of these features to achieve high-rate, timely and energy-efficient processing.

In the first part of the thesis, we focus on high-end servers and utilize two main approaches to achieve high throughput processing: vectorization and thread parallelism. We employ vectorization for the case of pattern matching algorithms used in security applications. We show that re-thinking the design of algorithms to better utilize the resources available in the platforms they are deployed on, such as vector processing units, can bring significant speedups in processing throughout. We then show how thread-aware data distribution and proper inter-thread synchronization allow scalability, especially for the problem of high-rate network traffic monitoring. We design a parallelization scheme for sketch-based algorithms that summarize traffic information, which allows them to handle incoming data at high rates and be able to answer queries on that data efficiently, without overheads.

In the second part of the thesis, we target the intermediate tier of computing devices and focus on the typical examples of hardware that is found there. We show how single-board computers with embedded accelerators can be used to handle the computationally heavy part of applications and showcase it specifi-

cally for pattern matching for security-related processing. We further identify key hardware features that affect the performance of pattern matching algorithms on such devices, present a co-evaluation framework to compare algorithms, and design a new algorithm that efficiently utilizes the hardware features.

In the last part of the thesis, we shift the focus to the low-power, resource-constrained tier of processing devices. We target wireless sensor networks and study distributed data processing algorithms where the processing happens on the same devices that generate the data. Specifically, we focus on a continuous monitoring algorithm (geometric monitoring) that aims to minimize communication between nodes. By deploying that algorithm in action, under realistic environments, we demonstrate that the interplay between the network protocol and the application plays an important role in this layer of devices. Based on that observation, we co-design a continuous monitoring application with a modern network stack and augment it further with an in-network aggregation technique. In this way, we show that awareness of the underlying network stack is important to realize the full potential of the continuous monitoring algorithm.

The techniques and solutions presented in this thesis contribute to better utilization of hardware characteristics, across a wide spectrum of platforms. We employ these techniques on problems that are representative examples of current and upcoming applications and contribute with an outlook of emerging possibilities that can build on the results of the thesis.

Keywords: hardware-aware, parallelism, distributed processing, high-end, intermediate, resource-constrained

List of Publications

Included publications

1. **Charalampos Stylianopoulos**, Magnus Almgren, Olaf Landsiedel, Marina Papatriantafilou, "Multiple Pattern Matching for Network Security Applications: Acceleration through Vectorization," in *the Journal of Parallel and Distributed Computing (JPDC)*, vol. 137, pp. 34 - 52, Elsevier 2020.
   The above is an extended and elaborated version of the work that previously appeared in:
   Charalampos Stylianopoulos, Magnus Almgren, Olaf Landsiedel, Marina Papatriantafilou, "Multiple Pattern Matching for Network Security Applications: Acceleration through Vectorization," in *the Proceedings of the 46th International Conference on Parallel Processing (ICPP)*, pp. 472-482, IEEE 2017.

2. **Charalampos Stylianopoulos**, Ivan Walulya, Magnus Almgren, Olaf Landsiedel, Marina Papatriantafilou, "Delegation sketch: a parallel design with support for fast and accurate concurrent operations," in *the Proceedings of the 15th European Conference on Computer Systems (EuroSys)*, Article 4, pp. 1–16, ACM 2020.

3. **Charalampos Stylianopoulos**, Linus Johansson, Oskar Olsson, Magnus Almgren, "CLort: High Throughput and Low Energy Network Intrusion Detection on IoT Devices with Embedded GPUs," in *the Proceedings of the 23rd Nordic Conference on Secure IT Systems (NordSec)*, Secure IT Systems, pp. 87–202, LNCS vol. 11252, Springer 2018.

4. **Charalampos Stylianopoulos**, Simon Kindström, Magnus Almgren, Olaf Landsiedel, Marina Papatriantafilou, "Co-Evaluation of Pattern Matching Algorithms on IoT Devices with Embedded GPUs," in *the Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*, pp. 17–27, ACM 2019.

5. **Charalampos Stylianopoulos**, Magnus Almgren, Olaf Landsiedel, Marina Papatriantafilou, "Geometric Monitoring in Action: a Systems Perspective for the Internet of Things," in *the Proceedings of the IEEE 43rd Conference on Local Computer Networks (LCN)*, pp. 433-436, IEEE 2018.

6. **Charalampos Stylianopoulos**, Magnus Almgren, Olaf Landsiedel, Marina Papatriantafilou, "Continuous Monitoring meets Synchronous Transmissions and In-Network Aggregation," in *the Proceedings of the 15th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pp. 157-166, IEEE 2019.

Appended publication

1. **Charalampos Stylianopoulos**, Magnus Almgren, Olaf Landsiedel, Marina Papatriantafilou, Trevor Neish, Linus Gillander, Bengt Johansson, Staffan Bonnier,"Industry Paper: On the Performance of Commodity Hardware for Low Latency and Low Jitter Packet Processing," in *the Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems (DEBS)*, ACM 2020 (included as an Appendix).

Personal Contribution

I contributed to **Paper 1**, **Paper 2**, **Paper 5**, **Paper 6** and **Paper 7** as the lead designer and main implementer. I also led the writing of the manuscripts and collaborated with all other authors. In **Paper 2**, the design of Delegation Sketch was performed in collaboration with Iwan Walulya.

Paper 3 builds on a prototype implementation from Linus Johansson and Oskar Olsson. My contributions in this paper include: identifying the research problem, extending the implementation, designing and performing extensive benchmarks and leading the manuscript writing.

Paper 4 builds on an implementation by Simon Kindström. I extended that implementation by integrating PFAC and vectorized versions of DFC into the benchmark, introducing a new algorithm (HYBRID) and designing experiments to study its performance. I was also the main manuscript writer.

The work leading to **Paper 7** was performed during my research internship in Ericsson. I was the lead writer and main designer of the experiments in this paper and collaborated with all other authors.

Acknowledgments

I would like to start by thanking the people that made this thesis possible: my supervisors Marina Papatriantafilou, Magnus Almgren and Olaf Landsiedel. Caring supervisors are hard to find, and I was lucky to have three such people. They helped me in innumerable ways in every step of the way.

I am also grateful to the co-authors and collaborators that contributed to the work in this thesis. Special thanks to Ivan Walulya, Linus Johansson, Oskar Olsson and Simon Kindström. I would also like to thank the people that made my internship a fun and fruitful experience. Many thanks to Staffan Bonnier, Linus Gillander, Bengt Johansson, Trevor Neish, David Wahlstedt, Björn Svensson, Anders Fagerlind and Harald Lüning.

I am honored to have Prof. Angelos Bilas as the faculty opponent during the thesis defence. I would also like to thank the members of the grading committee: Prof. Dr. Jörg Keller, Assoc. Prof. Sabri Pllana, Prof. Herbert Bos and Prof. Ioannis Sourdis for participating in my defence. Many thanks to my examiner Aarne Ranta and the follow-up groups that supported me during my studies.

Special thanks to the department's administration that made many tasks easier for me: Eva Axelsson, Rebecca Cyren, Marianne Pleen-Schreiber, Clara Oders, Lars Norén and Michael Morin, as well as my manager Tomas Olovsson.

I am also grateful to past and present colleagues in the division that made Chalmers a fun place to work. Many thanks to Adones, Ali, Aljoscha, Amir, Aras, Bapi, Bastian, Bei, Beshr, Carlo, Christos, Dimitris, Elad, Fazeleh, Francisco, Georgia, Hannah, Yiannis, Iosif, Joris, Karl, Katerina, Nasser, Oliver, Paul, Philipas, Romaric, Thomas, Valentin T., Valentin P., Vincenzo and Wissam.

I would also like to thank the friends that were with me all this time. Thank you Petros, Vaggelis, Stavros, Angelos, Chloe, Maria, Vasiliki, Manos, Suvi, Nikos, Kostas, Efi, Elias, Dinos and Gesti.

My thanks go to my family, my parents and my siblings for their unconditional love and trust in me. I owe it all to them. Finally, many thanks to Kelly for her love, trust and friendship against all odds.

Charalampos Stylianopoulos
Göteborg, August 2020

Contents

Part I

Introduction

1 Thesis Overview

In the last two decades, cloud computing has become one of the most predominant and well-established computing paradigms. Computing resources, storage, network and software services are aggregated in data centers and offered on-demand to a vast number of users over the Internet. Cloud computing infrastructure is the driving force of today's major distributed systems such as those run by Google and Facebook and cloud computing has up to now appeared to be the solution to all the challenges any application might face: scalability, elasticity, connectivity, high performance, low cost, reliability and security.

Recently, this traditional computing paradigm is being challenged by two upcoming (and interconnected) trends that are quickly gaining ground. The *first major trend* is the introduction of connectivity and computing components to everyday objects, i.e., the Internet of Things (IoT). The number of devices that are expected to be connected to the Internet in the coming years is impressive [1], with 24.9 billion expected connected devices by 2025 [2]. This includes everyday objects, cars, as well as devices that are part of the electricity grid and the industry [3]. However, what is even more impressive is the amount of data that they will produce. As an example, a modern connected car is equipped with numerous sensors that are generating 4 TB of data per day [4], that cannot all be sent to the cloud for processing. The challenge associated with all that data is now this: *how, when and where* to process the high volumes of data, in order to

extract value [5] [6]? Moreover, among rising concerns regarding the security of that data from IoT and the devices that produce them [7], additional security countermeasures are often deployed. These countermeasures need to inspect and process the generated data (e.g. for intrusion detection) and have potential to become a processing bottleneck.

The *second major trend* stems from the next generation of industrial applications, a.k.a. Industry 4.0. Industrial automation systems are now gaining networking capabilities and become true cyber-physical systems, i.e., they combine physical components (e.g. actuators and motors) that gather data from sensors, with a cyber counterpart that is connected with other systems and controllers that take decisions on that data and send back commands to the physical counterpart [3]. As such, industrial automation systems are connected, with either wired or wireless connections, to each other and with the controller. They constantly generate and transfer data for analysis at a computing node. A typical example of such analysis is predictive maintenance [8], where sensor readings are used to predict the lifetime of machinery and schedule maintenance before a malfunction occurs, thus ensuring that there are no interruptions in the production line. In this example, the application needs to continuously monitor, either data from individual sensors, or combined information from a set of sensors and take action when unexpected variations are detected [9, 10].

The net result of these two trends is a new set of applications with a range of requirements, from low latency control loops that quickly react to new data, to high throughput processing of aggregated data coming from a large number of connected sources. When considering such applications, the traditional cloud computing paradigm is no longer a panacea. In the case of IoT applications, the sheer number of connected devices and amount of data generated by each device makes it infeasible to send all that traffic to the cloud, both in terms of network bandwidth and processing capacity. In the case of Industry 4.0 applications, the need to take control decisions quickly (within milliseconds [11], see also the Appendix) means that sending the data to the cloud, doing the processing there and then receiving back the results adds unnecessary delays that real-time applications cannot tolerate.

The aforementioned requirements call for timely processing of data close to its source in an efficient, distributed manner. For this reason, it is necessary to decentralize the processing that is typically done in the cloud and distribute it along different layers of computation, along the path of the data. This, in turn, requires *hardware-aware* processing algorithms that can be efficiently deployed at various tiers of computing infrastructure, taking into account the hardware capabilities and constraints. This thesis considers how algorithms and processing applications can be adapted to better utilize the architectural features found

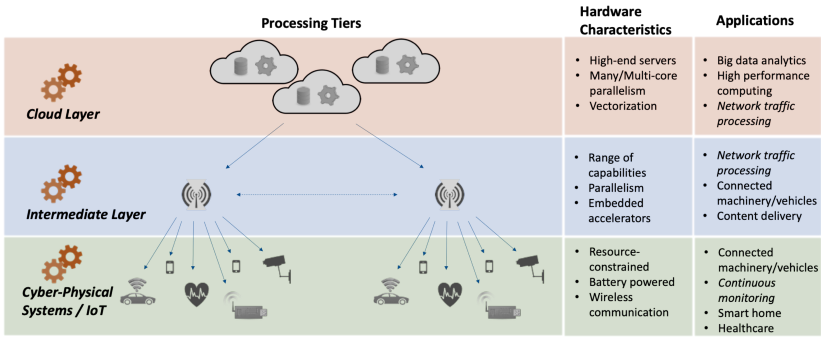


Figure 1.1: An example of processing across tiers of computing infrastructure. It includes a cloud layer with high-end servers, an intermediate layer close to the gateways and the IoT layer with resource-constrained devices.

across those tiers of computation. Next, we explain the opportunities and new challenges introduced by processing across a range of computing infrastructure.

1.1 Motivation

The core idea of computing across tiers of infrastructure is to *move the processing to where it is most needed*, closer to the data origin. In this paradigm, the processing and control logic that would typically be found on cloud servers are now pushed down to intermediate nodes, closer to the sources of data. Base stations and gateways will thus be enhanced with processing capabilities and storage. As such, they will be entrusted, e.g., with the processing of aggregated data coming from IoT networks, or some of the control logic of Industry 4.0 applications. Moreover, the IoT networks themselves will take over some of the processing and control logic, so that a significant portion of data does not have to be forwarded upwards. An example of such a tiered computing architecture is shown in Figure 1.1. By processing a large portion of the generated data close to the IoT layer, the up-link network bandwidth will no longer be a limiting factor and there is an opportunity to significantly reduce the overall processing latency.

Nowadays, we start seeing such aspects of computing in tiers of processing infrastructure and the way they affect different domains. Some characteristic examples are the following: (i) As mentioned above, existing and future Industry 4.0 applications, such as factory automation, machine-to-machine communication and autonomous driving require fast, real-time processing of the data they generate, which makes sending that data to the cloud infeasible. Instead,

processing is moved, on or close to the cyber-physical systems [12]. In many cases, the processing and the networking infrastructure co-exist on the same tier of computing architectures [13, 14]. (ii) In the automotive industry, several Electronic Control Units (ECUs) inside the car are being extended with more processing capacity, with similar hardware to what is typically common in data centers, including Graphic Processor Units (GPUs) [15]. These ECUs take over new processing tasks that range from sensor fusion to image processing, keeping the processing close to where the data is generated.

The concept of computation at different tiers of devices is being expressed through many proposed computing paradigms [16], such as fog computing, mobile edge computing and cloudlets. In fog computing [17], switches and access points in the network, as well as the IoT devices themselves take on processing tasks. In the mobile edge computing paradigm [13, 18], parts of the mobile broadband infrastructure become responsible for processing, using servers that are co-located with base stations (see also the Appendix). Cloudlets [19] are small scale, dedicated data centers that are deployed close to applications. The common premise in all those paradigms is that processing that is typically done in data centers, is now distributed and deployed on a wide range of devices.

The distribution of processing responsibilities to different tiers of platforms opens new, interesting research questions in several ways. On one hand, it comes with a new set of challenges, which mainly revolve around the problems of (i) how to distribute computational tasks on different tiers [12], (ii) how to move them there, (iii) in what ways the different components interact and connect with each other [5] and (iv) how to maintain Quality of Service [20]. On the other hand—and this is important in the context of this thesis—processing across different tiers of platforms brings together applications with different requirements that target different platforms, under the same computing approach. Processing methods originally designed for servers in the cloud now also become relevant at the intermediate layer and must adapt to the hardware found there. At the same time, computational tasks designed to operate on aggregated data on a single node can benefit if the processing logic is made distributed and handled *close to*, or even *by* the nodes that produce that data. This increases the design space of existing solutions and poses interesting research questions.

1.2 Hardware and Algorithm Diversity

As mentioned earlier, processing is no longer done only in massive servers but distributed across a wide spectrum, including intermediate processing layers (e.g. close to base stations) and resource-constrained devices (Figure 1.1). This

is challenging, since the hardware found at different layers is very diverse. Applications need to be *hardware-aware*, in order to take this into account and make the most out of the processing and communication capabilities at each layer.

In the cloud layer, the hardware typically consists of *massively parallel* servers with abundant storage and computing resources. Applications that are deployed at this layer must be adapted to take advantage of high parallelism and are usually oriented towards *high processing throughput*. We outline some of the techniques and challenges of efficient processing at this layer in Section 1.3.1.

The intermediate layer is typically flexible and in itself includes a range of devices. Some deployments at this layer have very similar characteristics to the cloud layer [19] and consist of powerful parallel servers. On other occasions, the hardware at this layer is more resource-constrained and strikes a balance between high performance and low energy consumption. Typical examples are *embedded, yet relatively powerful devices* such as clusters of Raspberry Pis and Odroid boards [21]. An interesting feature of this layer is that the hardware found here is rapidly evolving and some of the capabilities typically found in cloud servers [22], such as massive parallelism through general purpose *Graphics Processor Units (GPUs)*, have trickled down to this layer (see Section 1.3.2).

The cyber-physical/IoT layer is dominated by *resource-constrained devices*, typically sensors with very limited processing capabilities [23]. Those devices often have just enough power to collect data from their environment and send them to the upper layer. Many deployments in this layer rely on wireless, battery-powered devices that are expected to operate autonomously and without maintenance for months or years [24]. As such, energy consumption is a primary concern at this layer and applications need to optimize their processing and communication protocols accordingly. A summary of communication protocols with emphasis on energy consumption is presented in Section 1.3.3.

1.3 Background

As mentioned earlier, computing at a wide range of platforms is challenging due to the different nature and characteristics of the devices involved. In the previous section, we outline some of those hardware characteristics. In this section, we provide the background on techniques that make use of these characteristics at each layer of devices. Specifically, we discuss parallallelization and vectorization techniques used in high-end servers, accelerator techniques that are relevant at both high and intermediate tier of devices, as well as topics on wireless sensor networks that consist of low-end, resource-constrained devices.

1.3.1 Parallelism and Vectorization

In the last two decades, multi-core processors have become ubiquitous and have permeated the design and architecture of many devices, from smartphones and tablets to massive servers. This shift in processor design is directly related to the breakdown of Dennard's scaling [25]: the observation that the power consumption of transistors is proportional to their size. Near the year 2000, Dennard's scaling stopped being in effect, due to physical limits in heat dissipation and threshold voltage. As a result, semiconductor companies stopped trying to reduce the size of transistors as a way to increase the processing frequency of a single core. Instead, they started packing more cores per chip. Nowadays, a single chip may support multiple hundreds of cores [26]. These cores typically communicate with each other through shared memory.

This widespread adoption of multicores has brought a corresponding shift in software design. Application developers can no longer rely on processors becoming faster over the years. Instead, high performance must be gained through *parallelization*, in order to utilize all the available cores in the system. In practice, however, parallelization does not come for free and it is hard to achieve. Many applications are inherently sequential (e.g. a finite state machine traversal) or have parts that cannot be parallelized and must be performed in mutual exclusion with other threads (critical sections). Amdahl's law [27], used to predict the performance of parallel programs, suggests that even under highly parallel platforms with hundreds of cores, the performance gain due to parallelism on such applications is bound by the portions of their code that is sequential. In addition to that, co-ordination between cores, e.g. through shared data-structures, creates additional serialization points and points of contention that make parallelization a hard task. Often, hardware-aware algorithm designs are required to adapt applications to modern computing environments.

Apart from thread parallelism that utilizes multiple cores, modern processors also support *data parallelism* within each core. This type of parallelism is called Single Instruction Multiple Data (SIMD) because, at each step of the execution, a single instruction is issued that operates on multiple data. It is usually implemented by a separate hardware pipeline that supports (vector) registers and execution units that operate on arrays of values, rather than single values, with a single instruction. As an example, a typical addition operation that adds the values stored in two registers and stores the result in a third register, is extended to operate on two arrays of registers (e.g. 16 32-bit registers) and store the result in a third array. The cost of these vector instructions for addition is usually the same as their scalar counterparts.

SIMD parallelism was introduced as early as in 1966 [28] and became

widespread through its use in the Cray supercomputer [29] that used vectorization in the 1970s. Since then, vector execution units have been part of most mainstream processor designs [30, 31]. However, the usefulness of vectorization as a technique has been limited to applications that are amenable to data parallelism and requires that values must first be brought to the vector registers. If the application does not store the data that needs to be processed in consecutive memory locations and access it in a consecutive manner, that application cannot be vectorized, at least not efficiently. As a result, successful uses of vectorization have up to now been limited to specific, number-crunching applications with very simple and predictable access patterns.

In the last few years, vectorization is regaining importance and relevance. As the per-core frequency can no longer increase, vendors are using vectorization as a means for applications to reach higher performance [32]. This shift is mainly shown in two ways: (i) The number of values that can be stored in each vector register (vector length) increases. Starting from 128 bit long, vector registers increased within a few years to 256 bit and recently to 512 bit long vector registers [33]. This increased register size means that more data values can be operated, simultaneously, with a single instruction, which increases the benefits of using SIMD. (ii) New vector instructions. Modern vector instruction sets are enriched with support for more complex instructions. Notably, the *gather* instruction allows reading data from non-consecutive memory locations in memory and storing then in a vector register. Equivalently, the *scatter* instruction allows storing the contents of a vector register to non-consecutive memory locations. Hardware support for such instructions is a step towards lifting one of the main drawbacks in vectorizing applications and allows a wider range of applications to get performance benefits from vectorization.

In summary, parallelization and vectorization are techniques relevant to high-end servers. Later in this thesis, we describe how we use them for applications that require high-throughput processing.

1.3.2 General Purpose Computing in GPUs

Graphic Processor Units (GPUs) are hardware accelerators, originally designed for graphic operations such as rendering and ray tracing. They are highly parallel platforms, with thousands of processing elements per GPU, but each processing element usually runs at a slower frequency than a CPU thread and typically lacks the sophisticated mechanisms found in CPUs (e.g. out-of-order execution, branch prediction etc). The large number of processing elements make them a good fit for graphic and image processing applications, where usually every processing element is responsible for processing a single pixel.

Most GPUs are standalone co-processors, with their own separate memory (*device memory*), connected to the rest of the system via a data bus. The memory hierarchy of GPUs includes different types of memory [34], such as: (i) on-chip caches used by each processing element (*private memory*) (ii) shared caches between processing units (*local memory*), (iii) read-only memory for fast access to addresses with spacial locality (*constant memory*) and (iv) memory accessible for reads and writes by all processing elements (*global memory*). Another important characteristic of GPU architectures is the use of large register files that allow fast context-switching between processing tasks. This allows GPUs to hide long memory access latency by replacing a task that is waiting for a memory request to be served with another task that can do useful work, similar to some CPU designs [35]. Finally, the processing elements in most GPUs are organized in groups that issue instructions synchronously, in lock-step, with each element operating on a different part of the data, similar to SIMD vectorization.

In the last fifteen years, GPUs have been proven increasingly successful for many more applications than originally designed for. Their highly parallel nature makes them good alternatives to CPUs, especially for applications where threads can operate on disjoint parts of data without heavy inter-thread communication (also called embarrassingly parallel applications). As a result, they have been used extensively in image recognition, machine learning and many high-performance computing applications [36]. Another characteristic that makes them appealing alternatives to CPUs is their overall low power consumption per unit of computation [37].

Due to their applicability in various tasks, GPU platforms have evolved rapidly and have become a crucial part of many processing platforms. Lately, GPUs have permeated the intermediate layer of devices, as small, *embedded co-processors* in single-board computers [21]. Embedded GPUs at this layer offer less parallelism and processing power than their high-end counterparts, and are oriented towards lower energy consumption [38]. Their architecture has some interesting characteristics that distinguish them from high-end GPUs. As an example, embedded GPUs typically do not have a separate device memory and instead share the same physical memory with the CPU.

The use of GPUs for general purpose computing (GPGPU) has been made possible and widespread through the use of two programming frameworks: CUDA [39] and OpenCL [40]. CUDA was the first framework to be released and the one that popularized GPGPU computing, targeting primarily NVIDIA GPUs. OpenCL is an open standard and focuses on portability across different devices. Deploying an application on GPUs typically requires the following steps: (i) the data to be processed is copied from the CPU into the device memory of the GPU, (ii) the CPU enqueues a command to the GPU to begin processing the data

through a predefined program written for the GPU (kernel), and (iii) the GPU copies the results back to the main memory of the CPU.

Deploying efficient applications on GPUs is challenging due to several reasons. First, an existing parallel application usually needs to be re-written under a GPU programming framework. Second, the performance of the application often depends heavily on the hardware characteristics, such as the device memory size, the complex memory hierarchy and the number of threads that execute instructions synchronously (warp). Another issue that affects the performance of GPU applications is warp divergence, i.e., the situation where different threads in a warp must execute different parts of the code, e.g., due to an if statement. Finally, in many cases where the application requires frequent communication between the CPU and the GPU, the data bus between them is a performance bottleneck.

In summary, GPUs are widely used accelerator units for offloading general purpose computing tasks, but come with challenges on how to make use of their hardware features. Later in this thesis, we show how we make use of GPUs, in particular embedded GPUs.

1.3.3 Wireless Sensor Networks

On the low end of the spectrum of computing devices in Figure 1.1, the hardware characteristics are significantly different than in other layers. Here, the main hardware components are: (i) a set of simple sensory hardware that periodically collects data from the environment, (ii) a resource-constrained micro-controller unit (MCU) for simple data and packet processing, and (iii) a radio transceiver for communication with other nodes that is used to form networks (either structured or mesh).

Sensors are typically battery powered and are expected to operate without service for long periods of time. As such, the battery lifetime is the most valuable resource and the design of hardware and software for wireless sensors emphasizes on minimizing the energy footprint. As a consequence, applications on wireless sensors need to deal with the fact that processing components are very simple and *resource-constrained*.

In addition to computation, communication in Wireless Sensor Networks (WSN) is also challenging, due to its energy cost. In fact, the radio is one the most energy-hungry components, often consuming up to 10 times more energy than the MCU. For this reason, the goal of most communication protocols is *radio duty cycling* (RDC), where the radio is kept off as much as possible and is turned on for only a small fraction of the total time. A simple and commonly used RDC policy is to turn the radio on a fixed number of times per second,

called the *channel check rate* (CCR) [41]. A node that wants to transmit, will keep transmitting for a duration of at least $1/CCR$ seconds to ensure that all neighbouring nodes have a chance to turn their radio on and receive. The channel check rate is a tunable parameter of the protocol that directly affects the battery lifetime of the nodes.

In addition to energy consumption, latency and reliability are other important considerations in WSNs. Sensors usually form multi-hop networks over unreliable and lossy links that are constantly subject to interference. For this reason, a large body of WSN research has focused on how to design reliable and low-latency network protocols. Traditionally, wireless communication protocols attempt to avoid cases where packet transmission from multiple nodes overlaps in time, since this leads to interference and reduces the probability that other nodes in the network will correctly receive packets. Some recent protocol designs follow the opposite approach: transmissions from different nodes are tightly scheduled in time and start concurrently. Due to the *capture effect* [42], nodes in the network will successfully receive one of the packets if all transmissions start at the same time (e.g. within 160μsec for IEEE 802.15.4 technologies [43]) and one packet is at least 3dB stronger than the rest. Moreover, due to *constructive interference* [44], if two packet transmissions start within 0.5μsec from each other and the packets contain the same contents, these transmissions do not interfere with each other. Utilizing those two effects in wireless communications has given rise to protocols that achieve highly reliable, low latency communication [43–45].

In summary, in wireless sensor networks, energy consumption, latency and reliability are key objectives. Later in this thesis, we show how to design applications and protocols with these objectives in mind.

1.4 Representative Problems

In this section, we describe three representative applications that are used as a basis for the work in this thesis. These applications capture many of the challenges and requirements of data processing scenarios mentioned in the beginning of this introduction and are relevant for deployment on different layers of computing platforms described in Section 1.3. The applications we consider relate to *monitoring* data, coming from sources such as network packets or sensor readings. In this thesis, we focus on how the data processing involved in those monitoring applications can be done efficiently, on the hardware they are typically deployed on.

1.4.1 Network Intrusion Detection and Pattern Matching

In the first problem domain, we consider the problems and challenges involved in *pattern matching*, with a focus on its application in Network Intrusion Detection Systems (NIDSs). Pattern matching for NIDSs falls into the category of applications that require fast processing of data coming from multiple sources that is aggregated and processed on the high-end, or the intermediate layer of devices (Figure 1.1).

NIDSs are typically found at the entry point of networks and their purpose is to analyze the incoming and outgoing network traffic to detect any malicious behavior, ranging from unauthorized access, malware that exploits software vulnerabilities, data exfiltration, etc. They typically employ sophisticated analysis that considers not only the packet headers but also the contents of each packet (deep packet inspection [46]). There are many available NIDSs, with Snort [47] and Zeek (formerly called Bro [48]) being some of the most established and mature in the open-source community.

Network Intrusion Detection Systems gain new significance in the context of processing across a wide range of devices. We consider deployments of NIDS on: (i) high-end servers in the cloud and (ii) the intermediate layer of devices between the cloud and the IoT. On high-end servers, NIDSs are deployed to protect large networks with high volumes of network traffic. Nowadays, with the growth of Network Function Virtualization (NFV) technologies [49, 50], applications such as firewalls and NIDSs are moving away from custom hardware boxes and into the cloud, where they are deployed on general purpose servers as virtual functions. This deployment model offers flexibility, since virtual functions can be scaled up or down and replaced easily, depending on the traffic load.

On the lower end of the spectrum of devices, IoT networks are connected to the Internet, sending sensor readings upwards towards the cloud and receiving back control traffic. The end-devices producing sensitive data are potential attack targets. However, since they are typically resource-constrained, traditional security mechanisms cannot be easily deployed there. Hence, it is important to add protection mechanisms, both at the entry point of the network and along the data path towards the cloud. For this reason, we also focus on the deployment of NIDSs on the intermediate layer of devices, where the computational resources make it possible to deploy some security countermeasures to protect the vulnerable networks of resource-constrained devices.

An essential building block of many NIDSs is *pattern matching*, i.e., to discover if any of many predefined patterns exist in an input stream (*multiple*

pattern matching), for whitelisting or blacklisting.[1] Considering the processing involved in NIDSs, pattern matching is the most computationally intensive part and represents a major performance bottleneck. More than 70% of the running time of a NIDS can be spent on pattern matching [55, 56]. This fact, in conjunction with the ever-increasing rates of traffic that needs to be analyzed, pushes the performance of NIDSs to their limits. Achieving high pattern matching throughput is challenging yet crucial for these systems: if the processing throughput cannot match the incoming traffic rate, the system will have to start dropping packets and may miss potential attacks.

1.4.2 Network Monitoring and Sketches

In addition to Network Intrusion Detection, we also consider the processing involved in network traffic monitoring, with emphasis on high performance. Similarly to pattern matching, traffic monitoring also relates to high-speed processing at high-end or intermediate layer of devices (Figure 1.1).

Monitoring and maintaining statistics on network traffic is an important task that feeds valuable information to many other systems and actors. As an example, a NIDS may be interested in constantly maintaining information about the distribution of incoming traffic, since an abrupt change in that distribution may indicate an attack [57, 58]. Similarly, a controller in a Software Defined Network (SDN) may want to know the most popular destinations of traffic in the network, in order to perform dynamic flow scheduling [59]. Statistics extracted from data streams, such as top-k results, are useful in many scenarios [60].

In the context of different, interconnected layers of devices, traffic monitoring is challenging due to the high volume of traffic being generated, e.g. from IoT devices, aggregated at intermediate devices and sent to the cloud. Keeping up with the incoming traffic rate is a challenge that requires high throughput processing.

In addition, some traffic monitoring tasks come with inherent challenges in terms of memory consumption. As an example, consider a simple monitoring system that answers queries of the type: ”How many packets with source address X have entered the network?”. In order to be able to give an exact answer, at any point in time and for any given address, the system needs to store all the incoming addresses and their counts, which consumes memory proportional to the number of unique addresses found in the traffic stream. For high-speed

[1] Apart from its role in intrusion detection, pattern matching is also a core function in many other tasks, such as virus detection [51], text search [52] and genome analysis [53] [54].

networks with more than 10Gb Ethernet links, this approach quickly becomes impractical.

If an *approximate answer* to the aforementioned query is acceptable, there are algorithmic solutions using constant memory without storing the actual addresses. *Sketches* [61] are probabilistic data structures that are heavily used for the purpose of traffic monitoring because they offer a configurable trade-off between accuracy and memory consumption. They usually provide answers to queries that have error of at most ϵ with probability at least $1 - \delta$ (*probably approximately correct* [62]). Sketches typically use multiple hash functions to summarize the traffic stream using a fixed amount of space. The number of hash functions and the range of values of each hash function can be adjusted to offer accuracy guarantees on the queries performed on the sketch.

One of the challenges in creating new sketching approaches is to design new hashing techniques that improve the accuracy of the sketch while maintaining the same memory usage. Moreover, sketches need to be designed with high performance in mind. Specifically, due to the environments and the applications they are used in, sketches are required to have: (i) high insertion rate, so that they can keep up with the traffic rate of high-speed networks, and (ii) high query rate, i.e. the ability to support frequent queries on the sketch. The latter requirement is necessary for applications that continuously monitor traffic and have to react quickly to unpredictable changes [58].

1.4.3 Distributed Continuous Monitoring and IoT

Within the general area of monitoring applications, we shift the focus to resource-constrained IoT networks (Figure 1.1) and target the important problem of distributed monitoring of sensor readings, that is relevant to many applications that operate on data from IoT networks or industrial applications.

We address the issue of *continuously monitoring* a distributed set of sensor values and keeping track of a function of interest, defined over the network-wide aggregate of these values. Often, the goal is to always be able to detect whether the value of the monitored function has exceeded a predefined *threshold*. Keeping track of such a function is a basic building block for many IoT applications and control loops, e.g. for detecting outliers [63], hot-spots [64] or denial-of-service attacks [65].

Monitoring sensor values is a prime example of IoT applications that require local processing, close to the sources of data, in order to achieve timely monitoring and low latency detection of a threshold violation. Ideally, the monitoring logic can even be placed inside the IoT network and distributed to the sensor nodes themselves.

Keeping track of a function defined over a network-wide aggregate is a challenging task in practice. A simple solution is to aggregate every reading from every node in the network at a central entity and compute the aggregate there. Such an approach is impractical in networks with battery-constrained devices: using the radio for transmission or reception is the single most expensive operation in terms of energy [66]. Some solutions to the challenges associated with this problem are to compress the transmitted data [67, 68], or reduce the number of sensor readings that need to be transmitted, by letting all nodes locally determine whether a reading should be transmitted. However, finding such local criteria is challenging when the function to monitor is non-linear (e.g. the variance of the readings) yet it is non-linear functions that are particularly interesting for many real-world applications (e.g. detecting a denial-of-service attack, using the entropy of a series of readings [65]).

1.5 Related Work

In this section, we summarize related work that is relevant to hardware-aware algorithms, including each of the representative problems presented in Section 1.4. We also contrast related work against aspects that the state-of-the-art does not address, which later form the motivation behind the contributions included in this thesis.

1.5.1 Hardware-aware algorithm design

Algorithm engineering [69] is often viewed as a cycle between design, analysis, implementation and experimental evaluation of algorithms. In this cycle, the design of algorithms relies heavily on models that capture important characteristics of the hardware. Algorithms that are aware of the underlying hardware model and interact efficiently with it, have a clear benefit over algorithms (even asymptotically better ones) that do not take the hardware model into account.

Typical examples are hardware models that take into account the memory hierarchy (e.g. caches, main memory, disk) and algorithms that optimize their memory access pattern according to this hierarchy. Such models are used in e.g. block-based matrix multiplication that makes use of spatial cache locality [70], data structures such as B-trees [71] and many other applications [72]. Similar models also drive algorithmic engineering for algorithms that are deployed on specialized architectures, such as the Cell processor [73, 74] and Intel's SCC [75, 76] or similar platforms [77]. Recently, machine learning models have been proven successful in automatic software optimization (a part of algorithmic

engineering) based on the underlying hardware [78].

Hardware platforms often offer specialized resources e.g. support for vector instructions (Section 1.3.1) and GPUs (Section 1.3.2). Making use of such hardware resources is not always straightforward and requires design of algorithms that take them into account. Examples include redesigning database operations and data structures to make use of vectorization [79] or GPUs [80]. Moreover, acceleration using programmable networks cards [81] is starting to be used extensively in many applications [82, 83]. In all of the above cases, algorithms that successfully make use of the specialized hardware resources are specifically designed with the hardware capabilities in mind.

1.5.2 Pattern Matching and Hardware Characteristics

Pattern matching has been an active field of research for many years and there are many proposed approaches. The algorithm designed by Aho and Corasick [84] is one of the most well known and the one currently used by the Network Intrusion Detection System Snort. The first step of the Aho-Corasick algorithm is to create a finite-state automaton from a previously known set of patterns that belong to malicious attacks. Then, the algorithm scans the input traffic byte-by-byte to traverse the automaton, until it arrives at a final state that indicates the detection of an attack. Even though the Aho-Corasick algorithm performs only a small number of operations per byte, it fails to perform well in practice, due to poor cache locality. Nonetheless, the Aho-Corasick algorithm is a widely used pattern matching kernel, both in software and in hardware [85].

State-of-the-art approaches have been proposed to address the limitations of the Aho-Corasick algorithm. A family of algorithms in the literature replaces the state machine of the Aho-Corasick algorithm with filters. Choi et al. [51] use a series of succinct filters, created using a small part of each pattern that represents an attack. In this way, most of the benign input traffic is quickly filtered out, using cache-resident data structures. The part of the input that matches the information in the filters is further examined in a later verification phase that involves lookups in hash tables that contain the full patterns. Similarly, Moraru et al. [86] use a modification of Bloom filters [87] to scan both the input and the subset of patterns that are relevant. Sourdis et al. propose a similar selection of a relevant subset of patterns and implement it in hardware [88]. They also present a hardware implementation that uses perfect hashing [89] to find a potentially matching pattern for a given part of the input.

Motivating challenges for this thesis: Even though the state-of-the-art approaches have substantially increased the achieved throughput, they perform sub-optimally in modern architectures, because they fail to make use of the new

characteristics and features, as discussed in Section 1.3. On high-end servers, most pattern matching algorithms do not make use of the vector execution units and leave them underutilized. On intermediate layer devices, they fail to make use of unique architectural features, such as the embedded GPUs found at this layer. It is important to look deeper into the architecture of the devices NIDSs are deployed on and determine the features and characteristics that can be utilized to achieve high processing rates. We outline how the contributions in this thesis address those challenges on high-end servers in Section 1.7.1 and on intermediate layer devices in Section 1.7.2.

1.5.3 Sketches and Parallelism

There is a plethora of sketch-based algorithms that introduce different techniques to summarize data streams [61, 90]. The Count-Min Sketch [91] is a simple and widely used data structure that can answer approximate point queries, i.e. it returns the frequency of a certain key in the stream. It consists of d rows and w buckets per row, along with d different hash functions that map items into each row. Upon arrival of a new key (e.g. a new IP address), the key is separately hashed with each hash function and the value of the corresponding bucket is incremented by one. When querying for the frequency of a key, the key is hashed with all the hash functions and the answer to the query is the minimum value of the corresponding buckets. This value represents the closet approximation, since it is the bucket that has had fewer collisions with other, irrelevant keys. The approximation error of such a query is $\frac{e}{w}N$ with probability $1 - \frac{1}{e^d}$, where N is the number of keys that have been inserted in the sketch [91]. An invariant of the Count-Min Sketch is that the answer to a query is never less than the true frequency of the key in the stream. A different variation, the Count-Sketch [92] uses the median, rather than the minimum value as an estimator. Apart from those two approaches, the literature in sketches offers many other variations that focus on the accuracy/memory consumption trade-off [91–93].

Motivating challenges for this thesis: In order to match the processing rate required for high-speed networks, sketches need to be parallelized and make use of the underlying hardware features. However, most of the work proposed on sketches focuses on the single thread case and ignores parallelism. The few parallel designs that exist fail to achieve both high insertion and high query rate and focus on one of the two. We identify that there is a gap in the range of solutions for parallel sketches. We also show that the parallelization of a sketch has significant impact on its accuracy. There is great need for a sketch design that makes use of the platform's parallelism, handles millions of insertions per second and at the same time supports frequent, concurrent queries that return

accurate results. We outline how we address this need in Section 1.7.1.

1.5.4 Continuous Monitoring and IoT

Sharfman et al. [94] proposed a general method called *Geometric Monitoring (GM)* that can monitor any function (linear or not) defined over the average of network readings and keep track of its value with respect to a threshold. When using this algorithm, every node is capable of deriving constraints on its local values and avoid communication as long as those constraints are not violated. The GM method has been extended with sketches [95] and prediction models [96] and has been applied to outlier detection [63] and data stream queries [97].

In GM, any node must be able to let all other nodes in the network know that a local threshold has been violated and propagate the new estimate across the network, in an any-to-all communication pattern. However, most existing WSN protocols focus on either data collection (all nodes send data to a single sink) [68, 98] or data dissemination (the data from a single node is propagated across the network) [44].

Motivating challenges for this thesis: Apart from the existing general analysis of continuous monitoring algorithms, such as the one described above, the applicability to a real IoT deployment is unclear, from a practical perspective and the literature offers no insights in how the system aspects of IoT networks interact with such algorithms. Specifically, the underlying network stack can have a significant impact on the efficiency of the algorithm, in terms of energy consumption on the nodes, as well as latency and reliability of communication. Moreover, the resource-constrained nature of the sensor nodes makes the processing required by the algorithm challenging in practice. Finally, we note that existing network protocols are not designed for the communication pattern that is typically found in Geometric Monitoring (any-to-all) communication.

Due to the aforementioned gap between the application (continuous monitoring) and the network stack, it is important to study the behaviour of the application on top of network stacks in real-world sensor devices and identify the bottlenecks due to communication and processing. Additionally, finding ways of co-designing the network stack and the application can bridge this gap. Since existing network protocols are not a good fit for this application, it is necessary to re-design them or extend them and optimize their energy consumption with respect to the communication pattern of the application. We show how we approach those challenges in Section 1.7.3.

1.6 Research Questions

In this thesis, we study the core processing algorithms of characteristic applications that can be deployed in a wide spectrum of devices. For these algorithms we propose, implement and evaluate new designs that are closely coupled with the characteristics and capabilities of the hardware typically found across the different layers of computing devices.

By doing so, we attempt to demonstrate the following: Processing applications can greatly benefit (in terms of higher processing throughput and better energy efficiency) from algorithm engineering that provides tailored solutions that are aware of, and efficiently utilize, the hardware capabilities and characteristics of the platforms where these applications are deployed.

The work in this thesis investigates the following research questions.

- RQ1: How can hardware support be used to achieve high processing throughput on high-end and intermediate-layer devices?
- RQ2: How to co-design algorithms and communication protocols in order to better serve the needs of distributed applications?
- RQ3: How can algorithms that are traditionally viewed on one tier of computing devices be re-designed and deployed on different tiers?
- RQ4: How do the capabilities and system aspects of the platforms found in different computing layers affect the design and implementation of efficient processing algorithms?

RQ1 is relevant for applications that process large volumes of data, arriving at *high rates*. Processing kernels for network traffic processing are good such examples, including Network Intrusion Detection and packet monitoring, as described in Sections 1.4.1 and 1.4.2. In such applications, the requirement for high processing throughput often comes from the fact that the rate of processing needs to match the incoming data rate. In the pursuit of high processing throughput, the hardware itself offers capabilities that the processing algorithm must utilize. Such suitable hardware capabilities can be found both on the high-end and the intermediate layer (Figure 1.1). In the next section, we describe how we make use of vectorization, parallelism and hardware acceleration in traffic processing applications.

RQ2 becomes particularly relevant on the low tier of processing architectures in Figure 1.1, that consists of *resource-constrained*, connected devices, such as in Wireless Sensor Networks (Section 1.3.3). In this setting, the underlying communication protocol plays an important role in the performance of distributed

applications and is an important consideration when designing algorithms that are expected to perform efficiently (e.g., in terms of low latency and energy consumption). In the next section, we show how co-designing the application and the protocol leads to efficient distributed algorithms that perform well in realistic deployments.

RQ3 is relevant in the context of moving processing away from the cloud and distributing it in different tiers of devices (Section 1.1). When doing so, algorithms that were originally designed to operate on a specific tier of devices (e.g. the cloud) are now deployed in different tiers (e.g. on intermediate layer devices). This raises interesting research questions on how the performance of those algorithms will be affected by the change in hardware capabilities (e.g. from parallel servers to single-board computers) and the different communication environment (e.g. from data-center networks to WSN).

RQ4 is at the core of this thesis and is relevant to all tiers of processing architectures. At each tier, a way to meet the application's requirements (e.g. for high processing throughput, low latency, low energy consumption) is through algorithm designs that are tailored for the hardware platforms that they are deployed on. Hardware features and constraints are often the main factors that dictate the applicability and performance of algorithms. Algorithms can benefit greatly when they are redesigned in a way that takes into account the underlying platform. The work presented in this thesis includes examples of how such *hardware-aware* algorithm engineering can be done for various applications and platforms, as well as the benefits it brings.

We relate back to these research questions and how we address them in this thesis, in the context of the research contributions described next.

1.7 Thesis Contributions

The contributions in this thesis are solutions to the open problems discussed in Section 1.5 and connect to the research questions raised in Section 1.6. Table 1.1 shows which research questions are addressed in each part and chapter in this thesis, starting after the present chapter. We summarize these contributions below.

1.7.1 Parallel Data Processing on High-End Servers

In Part II of the thesis, we target the processing performed on high-end servers, through two representative applications: (i) pattern matching for intrusion detection and (ii) network traffic monitoring. The common challenge that we address

	Part II Massively Parallel Servers		Part III Embedded Accelerators		Part IV Resource-Constrained Devices	
	Chapter 2	Chapter 3	Chapter 4	Chapter 5	Chapter 6	Chapter 7
RQ1	●	●	●	●	○	○
RQ2	○	○	○	○	●	●
RQ3	○	○	●	●	●	●
RQ4	●	●	●	●	●	●

Table 1.1: Research questions addressed in each chapter in this thesis.

in both target applications is to make the most out of the parallelism that the hardware platforms at this tier of processing offer. By doing so, we contribute towards RQ1 and RQ4 and show: (i) how hardware features can be utilized to achieve high processing throughput and (ii) how can processing algorithms be redesigned to make efficient use of such features.

(A) Vectorized pattern matching

In Chapter 2 we target the data parallelism within each core at high-end servers and propose *V-PATCH*, a pattern matching algorithm that relies on vectorization to process multiple bytes of input, in parallel. This work builds on the observation that recent approaches to pattern matching that rely on quick filtering of the input, have brought the problem close to the processor and achieve good cache locality. As a result, long memory latency is no longer the dominant bottleneck and the computational part of pattern matching becomes significant. With that in mind, we target this computational part and show how to improve it further, through vectorization (see Section 1.3).

We follow a two-step approach. First, we propose a refined and extended filtering strategy that: (i) performs filtering based on cache-resident data structures and is effective for the types of patterns found in real traffic, and (ii) is simple enough to allow efficient vectorization. As an example, we deal separately with small, but frequently found patterns and perform more targeted filtering for longer patterns. Second, we design a vectorized version that uses specialized instructions to parallelize the computation performed on the filters, together with optimizations (e.g. filter merging) that allow us to make the most of the filtering design. We complement the design of both the scalar and the vectorized versions with an analytical model that predicts their respective performance based on the number of patterns inserted in the filters.

We evaluate the effectiveness of *V-PATCH* using real malicious patterns from Snort [99], against both real and synthetic traffic mixes. The results on two

platforms, an Intel Haswell processor and an Intel Xeon Phi co-processor, show up to 1.8x and 3.6x times faster processing throughput respectively, compared with the state-of-the-art. Furthermore, we find that the vectorized approach retains a stable speedup of 1.4x over the scalar one, as the number of malicious patterns increases. We also show that our analytical model that predicts the behaviour of our algorithms is very close to the experimental results on real data. Finally, we evaluate the performance of *V-PATCH* when deployed across many cores on a highly parallel platform and show that it achieves processing throughput of up to 45Gbps and close to 2 times higher throughput than the standard algorithm in the field.

(B) Parallel approximate traffic monitoring

In Chapter 3 we target the problem of traffic monitoring with sketches (see Section 1.5.3), in the context of highly parallel platforms. We propose *Delegation Sketch*, a parallel design for sketch-based algorithms that supports concurrent insertions and queries, at high rates. At the same time, we show that the queries on the sketch have better accuracy than some of the baselines, while maintaining the same memory consumption.

Our design is built around two techniques: (i) Domain Splitting and (ii) Operation Delegation. Domain Splitting is used to split the range of incoming keys to groups and assign each group to a different sketch. All occurrences of a key will be inserted only in the sketch assigned for that key. By doing so, we ensure that query operations are served fast: one needs to only search a single sketch for a key. This also makes queries more accurate, as we do not accumulate the errors of individual sketches. Operation Delegation is a mechanism that ensures efficient synchronization of operations from different threads on the same sketch.

Our design makes use of filters, i.e., small buffers that aggregate multiple occurrences of the same key. Their purpose is dual: (i) they allow threads to perform most of the insertions on filters rather than on the full sketch, which is faster due to their small size and (ii) they serve as a unit of synchronization between threads. Once a filter is full, it is handed over to a thread that is responsible for inserting the keys in that filter into the sketch. Finally, we also propose a "query squashing" optimization that aggregates queries on the same key by different threads into a single logical operation on the sketch, thus reducing the cost of queries, especially under high contention.

We evaluate the performance of *Delegation Sketch* across different dimensions: scalability, query rate, accuracy and processing latency. Our experiments with real and synthetic data on massively parallel platforms with up to 288 threads show that our approach supports up to 4X higher processing throughput and

performs queries with up to 2.25X lower latency than the next best-performing alternative. At the same time, *Delegation Sketch* has the same accuracy as the most accurate alternative, using the same amount of memory.

1.7.2 Fast and Energy-Efficient Processing on Embedded Accelerators

In Part III of this thesis, we target the intermediate tier of devices and the hardware found there. Specifically, we focus on single-board computers with embedded accelerators (GPUs) and we make use of their architectural features to accelerate the processing involved in Network Intrusion Detection Systems. The work in this part of the thesis contributes to RQ1, RQ3 and RQ4, by investigating the design of high-throughput algorithms on single-board computers (rather than high-end servers) and the effect of hardware features (embedded GPUs) on the design and performance of these algorithms.

(A) Integrating GPU Computing as Part of NIDS

In Chapter 4, we propose *CLort*, an extension to the widely used NIDS Snort [47], that is designed to make the most out of devices with embedded GPUs to accelerate pattern matching. Our work relies on the fact that the hardware at the intermediate layer, which serves as a gateway to IoT networks, is constantly being improved and it gains some of the architectural features that are so far used in more powerful platforms.

We analyze Snort and show how to offload the computationally heavy part of intrusion detection (pattern matching) into the GPU and integrate GPU computing into Snort's processing pipeline. We also integrate optimizations that allow this offloading to be performed in an efficient way, by overlapping the processing on both the CPU and the GPU.

Our evaluation shows that embedded GPUs are effective hardware platforms for Network Intrusion Detection at the intermediate tier of devices. We experiment with realistic traffic and show that using the GPU leads to 52% faster processing than the CPU, while consuming 32% less energy. By making the most of the available hardware resources, we show that single-board computers with embedded GPUs can be deployed as "security boxes" that are able to process traffic at high rates.

(B) Evaluation of Pattern Matching on Embedded GPUs

In Chapter 5, we take a closer look at the performance of pattern matching algorithms on embedded GPUs. Specifically, we present the results of an evaluation of different algorithms and investigate the particular characteristics of embedded GPUs that affect the performance of these algorithms. Based on the results of this evaluation, we also propose a new hybrid algorithm that combines ideas from the existing approaches and outperforms them.

The purpose of our benchmark is to establish a co-evaluation framework for different algorithms. We include existing algorithms that capture the main approaches on pattern matching, including a GPUs adaptation of our own, as well as a new, hybrid algorithm. We also examine a series of hardware characteristics that make embedded GPUs distinct from the regular, high-end GPUs, such as the lack of separate device memory.

We evaluate the included algorithms under our common benchmark, using a series of real data and malicious patterns extracted from Snort. We find that GPUs are attractive alternatives to CPUs for pattern matching at the intermediate layer of devices, in terms of processing throughput and energy efficiency. We also show that the unique features of embedded GPUs have a significant effect on the performance of pattern matching algorithms and need to be taken into account in the design of new approaches.

1.7.3 Distributed Processing on Resource-Constrained Devices

In Part IV of this thesis, we target networks of resource-constrained devices and study applications that distribute the processing to the cyber-physical devices themselves. Specifically, we focus on the application of continuous monitoring. We contribute towards RQ2, RQ3 and RQ4 by showing: (i) the effect that the underlying network protocol has on the design and performance of continuous monitoring, and (ii) the benefits of co-designing the protocol together with the application.

(A) A Full-System Perspective for Geometric Monitoring

In Chapter 6 of this thesis, we study Geometric Monitoring [94] (see Section 1.4.3) from a full-system perspective, when applied on real IoT networks. We design and deploy Geometric Monitoring on top of a mainstream wireless sensor network stack. Then, we thoroughly evaluate the performance benefits achieved in practice, the run-time behavior of the algorithm and the effects of packet losses.

We design the system on top of multi-hop mesh networks, without the need for maintaining a topology. When a node detects significant changes in its sensor readings, it will trigger a network-wide broadcast and inform every other node of its new value. This is done by network flooding, where a node that receives "new" information will broadcast it further to its neighbors. In the event that a node fails to receive an update by any of its neighbors, that node will be *out of sync* until a subsequent broadcast from the same origin arrives.

We investigate the important parameters of the network stack that determine the effectiveness of the algorithm in practice. As we show in this chapter, the rate at which nodes wake-up to receive traffic (Channel Check Rate, CCR) greatly affects the energy savings of the GM method. We evaluate our design using both full-system simulations and real IoT testbeds. Overall, we find that GM brings significant benefits to monitoring tasks, in terms of communication reduction. Specifically, when monitoring the variance and the average of real temperature data, GM achieves 3x and 11x reduction in duty-cycle, respectively. However, these benefits are limited compared to the communication reduction of the algorithm in isolation (4.3x and 44x respectively), due to baseline energy overhead of the network stack. Closer looks into the run-time behavior of the algorithm show that (i) the communication pattern varies greatly, and (ii) packet losses greatly impact the time a node is *out of sync* and reduce the ability of the algorithm to detect violations in a timely manner.

(B) Application and Protocol Co-Design for Geometric Monitoring

The findings from the work presented in Chapter 6 show that the network protocol used for communication between the nodes is a major prohibiting factor that limits the effectiveness of the Geometric Monitoring method. In Chapter 7, we build on this observation and take one step further: we co-design Geometric Monitoring with a modern network protocol. By doing so, we provide a practical realization of a system that continuously monitors sensor values with a high degree of communication suppression (due to properties of Geometric Monitoring) and operates at low duty-cycle with high reliability (due to properties of the underlying communication protocol).

As a starting point, we use a modern, low-power communication protocol (Crystal [98]) that relies on synchronous transmissions for fast and reliable data collection to a single sink. We extend this protocol to match the communication pattern of Geometric Monitoring which requires any-to-all communication. Moreover, we augment our design further by introducing an in-network aggregation technique, named *Arctium*, that leverages latent opportunities of the communication protocol to reduce the overall communication cost. Finally, we

redesign the application to better match the challenges related to processing in resource-constrained devices. Specifically, we identify that Geometric Monitoring requires every node to periodically check their local threshold violation criteria, which can be computationally challenging for resource-constrained devices. To this end, we propose a simple relaxation of the violation check that introduces a trade-off between computational efficiency and communication reduction.

Our experiments on both simulations as well as deployments with real IoT testbeds and data-sets, show that our design leads to an up to 10x reduction in energy consumption compared to previous work. Our in-network aggregation technique (*Arctium*) further reduces it by 1.13-1.38x. These results motivate that careful co-design of the application and the communication protocol can lead to low-power, continuous monitoring designs that have potential to perform efficiently on real world deployments.

1.8 Conclusions and Emerging Future Directions

In this thesis, we show that algorithm engineering, that emphasizes on utilizing the architectural features of hardware platforms, is crucial for the design of data processing algorithms that scale, maintain high processing throughput and are energy-efficient. We demonstrate this on the full spectrum of platforms, from highly parallel servers, to intermediate-tier devices and resource-constrained nodes, and in a variety of data processing applications.

At each of the three tiers of devices, we focus on key architectural features and show how to utilize them to improve the efficiency of applications. For the case of high-end servers, we focus on two key features: (i) vectorization, where we make use of dedicated vector instructions that allow us to uncover data parallelism in applications that would have otherwise not made the best use of the hardware platform and (ii) many-core parallelism, where we focus on data distribution and inter-thread coordination techniques that ensure scalability on massively parallel platforms, while preserving key correctness properties [62]. On the intermediate tier of devices, we make use of the interesting mix of platforms found there, specifically single-board computers with embedded accelerators. On such devices, we show how to offload computationally heavy tasks and identify how the unique characteristics of these platforms affect the performance of applications. Finally, on resource-constrained devices, we identify the key role of the interplay between the processing application and the communication protocol and propose algorithmic designs that integrate the two, on modern and realistic deployments.

In this thesis, we focus on applications that facilitate processing on a wide range of platforms and provide techniques to improve their performance, based on the hardware's features. We chose these applications as characteristic examples of the processing involved in these layers, but it would be interesting to study to what extent our techniques can be used in other applications. As the fields of IoT and Industry 4.0 grow (see also the Appendix) and new application domains emerge, it would be interesting to identify the main computational kernels that are the same across applications and study how they interact with the underlying hardware and the spectrum of the consistency levels that is achievable and useful.

As the hardware across different layers of devices is evolving, it is important to understand the change in types of hardware and architectural features. Future directions include insights on more hardware types that constitute a smaller, albeit important share of the platforms that are relevant across many tiers. As an example, Field-Programmable Gate Arrays (FPGAs) are configurable hardware platforms where the processing hardware itself and the way it interconnects can be programmed to complete a specific task, with inherently high parallelism. Such devices have already been proven useful in high throughput applications, e.g. pattern matching [89]. It would be interesting to see the role that FPGAs will play in the spectrum of computing infrastructure, as well as to determine how the techniques we present in this thesis can be expressed in FPGA based solutions.

Finally, an important step forward is to study how the techniques presented in this thesis can be integrated into systems that span across multiple tiers of processing hardware and interact with each other. Many applications, such as industrial automation and predictive maintenance have components that span across all tiers of infrastructure: sensor data collection from resource-constrained devices, prediction algorithms close to sources of data and data analytics in the cloud. The coordination and integration of all these components, each with its own architectural features and opportunities is a new exciting challenge that is relevant in the era of parallel and distributed computing over a range of platforms.

Bibliography

[1] D. Evans, "The Internet of Things: How the next evolution of the internet is changing everything," Cisco White Paper `https://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf`, January 2011, Accessed: 2020-06-17.

[2] P. Jonsson, S. Carson, G. Blennerud, J. Kyohun Shim, B. Arendse, A. Husseini, P. Lindberg, and K. Öhman, "Ericsson Mobility Report," `https://www.eric`

sson.com/4acd7e/assets/local/mobility-report/documents/2019/emr-november-2019.pdf, November 2019, Accessed: 2020-04-27.

[3] N. Jazdi, "Cyber physical systems in the context of industry 4.0," in *2014 IEEE International Conference on Automation, Quality and Testing, Robotics*, May 2014, pp. 1–4.

[4] B. Krzanich, "Data is the New Oil in the Future of Automated Driving," https://newsroom.intel.com/editorials/krzanich-the-future-of-automated-driving, November 2016, Accessed: 2020-04-27.

[5] M. Chiang and T. Zhang, "Fog and IoT: An Overview of Research Opportunities," *IEEE Internet of Things Journal*, vol. 3, no. 6, pp. 854–864, Dec 2016.

[6] M. Brodie, "Data: The World's Most Valuable Resource," Lectures in Computer Science on Big Data and Applications https://michaelbrodie.com/talks, July 2017.

[7] D. E. Sanger and N. Perlroth, "A New Era of Internet Attacks Powered by Everyday Devices," https://nytimes.com/2016/10/23/us/politics/a-new-era-of-internet-attacks-powered-by-everyday-devices.html, 2016, Accessed: 2020-06-17.

[8] D. Wu, S. Liu, L. Zhang, J. Terpenny, R. X. Gao, T. Kurfess, and J. A. Guzzo, "A fog computing-based framework for process monitoring and prognosis in cyber-manufacturing," *Journal of Manufacturing Systems*, vol. 43, pp. 25 – 34, 2017.

[9] S. Rangwala and D. Dornfeld, "Sensor Integration Using Neural Networks for Intelligent Tool Condition Monitoring," *Journal of Engineering for Industry*, vol. 112, no. 3, pp. 219–228, 08 1990.

[10] M. Gabel, A. Schuster, and D. Keren, "Communication-efficient distributed variance monitoring and outlier detection for multivariate time series," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 37–47.

[11] J. Sachs, G. Wikstrom, T. Dudda, R. Baldemair, and K. Kittichokechai, "5G radio network design for ultra-reliable low-latency communication," *IEEE network*, vol. 32, no. 2, pp. 24–31, 2018.

[12] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos, "Challenges and opportunities in edge computing," in *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, Nov 2016, pp. 20–26.

[13] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing—a key technology towards 5G," *ETSI white paper*, vol. 11, no. 11, pp. 1–16, 2015.

[14] Y. Ku, D. Lin, C. Lee, P. Hsieh, H. Wei, C. Chou, and A. Pang, "5G radio access network design with the fog paradigm: Confluence of communications and computing," *IEEE Communications Magazine*, vol. 55, no. 4, pp. 46–52, 2017.

[15] NVIDIA, "Volvo Selects NVIDIA DRIVE for Production Cars," https://nvidianews.nvidia.com/news/volvo-selects-nvidia-drive-for-production-cars, Accessed: 2020-05-19.

[16] K. Dolui and S. K. Datta, "Comparison of edge computing implementations: Fog computing, cloudlet and mobile edge computing," in *2017 Global Internet of Things Summit (GIoTS)*, 2017, pp. 1–6.

[17] L. Vaquero and L. Rodero-Merino, "Finding your way in the fog: Towards a comprehensive definition of fog computing," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 5, pp. 27–32, Oct. 2014.

[18] M. T. Beck, M. Werner, S. Feld, and S. Schimper, "Mobile edge computing: A taxonomy," in *Proc. of the Sixth International Conference on Advances in Future Internet*. Citeseer, 2014, pp. 48–55.

[19] T. Verbelen, P. Simoens, F. De Turck, and B. Dhoedt, "Cloudlets: Bringing the cloud to the mobile user," in *Proceedings of the third ACM workshop on Mobile cloud computing and services*, 2012, pp. 29–36.

[20] S. Taherizadeh, A. C. Jones, I. Taylor, Z. Zhao, and V. Stankovski, "Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review," *Journal of Systems and Software*, vol. 136, pp. 19 – 38, 2018.

[21] Hardkernel, "Odroid XU4," https://www.hardkernel.com/shop/odroid-xu4-special-price, Accessed: 2020-06-17.

[22] E. Kanellou, N. Chrysos, S. Mavridis, Y. Sfakianakis, and A. Bilas, "Gpu provisioning: The 80-20 rule," in *European Conference on Parallel Processing*. Springer, 2018, pp. 352–364.

[23] MEMSIC, "TelosB Mote Platform," http://www.memsic.com/userfiles/files/Datasheets/WSN/telosb_datasheet.pdf, Accessed: 2020-05-19.

[24] M. Ceriotti, M. Corrà, L. D'Orazio, R. Doriguzzi, D. Facchin, G. P. Jesi, R. L. Cigno, L. Mottola, A. L. Murphy, M. Pescalli, et al., "Is there light at the ends of the tunnel? Wireless sensor networks for adaptive lighting in road tunnels," in *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks*. IEEE, 2011, pp. 187–198.

[25] R. H. Dennard, F. H. Gaensslen, Y. Hwa-Nien, V. Leo Rideovt, E. Bassous, and A. R. Leblanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[26] Intel, "Intel Xeon Phi Processor," https://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-processor-product-brief.html, Accessed: 2020-06-17.

[27] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[28] Y. Oyanagi, "Future of supercomputing," *Journal of Computational and Applied Mathematics*, vol. 149, no. 1, pp. 147 – 153, 2002, Scientific and Engineering Computations for the 21st Century - Methodologies and Applications Proceedings of the 15th Toyota Conference.

[29] G. Bell, "A Brief History of Supercomputing:"the Crays", Clusters and Beowulfs, Centers. What Next?," `http://gordonbell.azurewebsites.net/supers/supercomputing-a_brief_history_1965_2002.htm`, 2002, Accessed: 2020-08-17.

[30] G. Mitra, B. Johnston, A. P. Rendell, E. McCreath, and J. Zhou, "Use of SIMD Vector Operations to Accelerate Application Code Performance on Low-Powered ARM and Intel Platforms," in *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, 2013, pp. 1107–1116.

[31] Intel, "Intrinsics for Intel Advanced Vector Extensions 2 (Intel AVX2) Instructions," `https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-intel-advanced-vector-extensions-2.html`, Accessed: 2020-06-17.

[32] Intel, "Vectorization: A Key Tool To Improve Performance On Modern CPUs," `https://software.intel.com/content/www/us/en/develop/articles/vectorization-a-key-tool-to-improve-performance-on-modern-cpus.html`, Accessed: 2020-05-19.

[33] J. Hofmann, J. Treibig, G. Hager, and G. Wellein, "Comparing the performance of different x86 SIMD instruction sets for a medical imaging application on modern multi- and manycore chips," in *Proc. of the 2014 Workshop on Programming Models for SIMD/Vector Processing*. 2014, ACM.

[34] A. Munshi, "The opencl specification," in *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE, 2009, pp. 1–314.

[35] J. Keller, W. J. Paul, and D. Scheerer, "Realization of prams: Processor design," in *Distributed Algorithms*, Gerard Tel and Paul Vitányi, Eds., Berlin, Heidelberg, 1994, pp. 17–27, Springer Berlin Heidelberg.

[36] NVIDIA, "Record 136 NVIDIA GPU-Accelerated Supercomputers Feature in TOP500 Ranking," `https://blogs.nvidia.com/blog/2019/11/19/record-gpu-accelerated-supercomputers-top500/`, Accessed: 2020-05-19.

[37] S. Huang, S. Xiao, and W. Feng, "On the energy efficiency of graphics processing units for scientific computing," in *2009 IEEE International Symposium on Parallel Distributed Processing*, 2009, pp. 1–8.

[38] ARM, "ARM Mali-T628 product page," `https://www.arm.com/products/multimedia/mali-cost-efficient-graphics/mali-t628.php`, Accessed: 2018-03-14.

[39] Nvidia, "About CUDA," `https://developer.nvidia.com/about-cuda`, Accessed: 2018-03-11.

[40] Khronos Group, "OpenCL Overview," `https://www.khronos.org/opencl/`, Accessed: 2018-03-11.

[41] A. Dunkels, "The ContikiMac radio duty cycling protocol," Tech. Rep. 2011:13, ISSN 1100-3154, 2011, Swedish Institute of Computer Science.

[42] K. Leentvaar and J. Flint, "The capture effect in FM receivers," *IEEE Transactions on Communications*, vol. 24, no. 5, 1976.

[43] O. Landsiedel, F. Ferrari, and M. Zimmerling, "Chaos: Versatile and efficient all-to-all data sharing and in-network processing at scale," in *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*. November 2013, ACM.

[44] F. Ferrari, M. Zimmerling, L. Thiele, and O. Saukh, "Efficient network flooding and time synchronization with Glossy," in *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks*, April 2011.

[45] T. Istomin, M. Trobinger, A. L. Murphy, and G. P. Picco, "Interference-resilient ultra-low power aperiodic data collection," in *Proceedings of the 17th ACM/IEEE International Conference on Information Processing in Sensor Networks*, 2018.

[46] P. Lin, Y. Lin, Y. Lai, and T. Lee, "Using string matching for deep packet inspection," *Computer*, vol. 41, no. 4, 2008.

[47] M. Roesch, "Snort - Lightweight Intrusion Detection for Networks," in *Proc. of the 13th USENIX Conf. on System Administration*, Seattle, Washington, 1999, USENIX Association.

[48] V. Paxson, "Bro: a System for Detecting Network Intruders in Real-Time," *Computer Networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999.

[49] R. Mijumbi, J. Serrat, J. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, 2015.

[50] Y. Li and M. Chen, "Software-defined network function virtualization: a survey," *IEEE Access*, vol. 3, 2015.

[51] B. Choi, J. Chae, M. Jamshed, K. Park, and D. Han, "DFC: Accelerating string pattern matching for network applications," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, Santa Clara, CA, 2016, pp. 551–565, USENIX Association.

[52] G. Navarro, "NR-grep: a fast and flexible pattern-matching tool," *Software: Practice and Experience*, vol. 31, no. 13, pp. 1265–1312, 2001.

[53] G. Navarro and M. Raffinot, *Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences*, Cambridge University Press, 2002.

[54] S. Memeti and S. Pllana, "Combinatorial optimization of DNA sequence analysis on heterogeneous systems," *Concurr. Comput. Pract. Exp.*, vol. 29, no. 7, 2017.

[55] S. Antonatos, K. Anagnostakis, and E. Markatos, "Generating Realistic Workloads for Network Intrusion Detection Systems," *SIGSOFT Softw. Eng. Notes*, vol. 29, pp. 207–215, 2004.

[56] J. B. D. Cabrera, J. Gosar, W. Lee, and R. K. Mehra, "On the statistical distribution of processing times in network intrusion detection," in *43rd IEEE Conf. on Decision and Control (CDC)*, Dec 2004, pp. 75–80.

[57] V. Gulisano, M. Callau-Zori, Z. Fu, R. Jiménez-Peris, M. Papatriantafilou, and M. Patiño-Martínez, "STONE: A streaming DDoS defense framework," *Expert Systems with Applications*, vol. 42, no. 24, pp. 9620 – 9633, 2015.

[58] Y. Li, R. Miao, C. Kim, and M. Yu, "Flowradar: A better netflow for data centers," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, Santa Clara, CA, Mar. 2016, pp. 311–324, USENIX Association.

[59] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proceedings of the Symposium on SDN Research*, New York, NY, USA, 2017, pp. 164–176, ACM.

[60] V. Gulisano, Y. Nikolakopoulos, I. Walulya, M. Papatriantafilou, and P. Tsigas, "Deterministic real-time analytics of geospatial data streams through scalegate objects," in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, New York, NY, USA, 2015, DEBS '15, p. 316–317, Association for Computing Machinery.

[61] G. Cormode, "Sketch techniques for approximate query processing," *Foundations and Trends in Databases. NOW publishers*, 2011.

[62] A. Rinberg and I. Keidar, "Intermediate value linearizability: A quantitative correctness criterion," 2020.

[63] S. Burdakis and A. Deligiannakis, "Detecting Outliers in Sensor Networks Using the Geometric Approach," in *2012 IEEE 28th International Conference on Data Engineering*, Washington, DC, USA, April 2012, pp. 1108–1119.

[64] I. Sharfman, A. Schuster, and D. Keren, "Aggregate threshold queries in sensor networks," in *2007 IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, March 2007, pp. 1–10.

[65] L. Feinstein, D. Schnackenberg, R. Balupari, and D. Kindred, "Statistical approaches to DDoS attack detection and response," in *Proceedings DARPA Information Survivability Conference and Exposition*, Washington, DC, USA, April 2003, pp. 303–314.

[66] A. Barberis, L. Barboni, and M. Valle, "Evaluating energy consumption in wireless sensor networks applications," in *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools*. IEEE, 2007, pp. 455–462.

[67] B. Havers, R. Duvignau, H. Najdataei, V. Gulisano, A. C. Koppisetty, and M. Papatriantafilou, "Driven: a framework for efficient data retrieval and clustering in vehicular networks," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, 2019, pp. 1850–1861.

[68] U. Raza, A. Camerra, A. L. Murphy, T. Palpanas, and G. P. Picco, "Practical Data Prediction for Real-World Wireless Sensor Networks," *IEEE Trans. on Knowledge and Data Eng.*, 2015.

[69] P. Sanders, *Algorithm Engineering – An Attempt at a Definition*, pp. 321–340, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[70] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick, "When cache blocking of sparse matrix vector multiply works and why," *Applicable Algebra in Engineering, Communication and Computing*, vol. 18, no. 3, pp. 297–311, 2007.

[71] M. A. Bender, E. D. Demaine, and M. Farach-Colton, "Cache-oblivious B-trees," in *Proceedings 41st Annual Symposium on Foundations of Computer Science*, 2000, pp. 399–409.

[72] L. Arge, M. A. Bender, E. Demaine, C. Leiserson, and K. Mehlhorn, "Abstracts collection – cache-oblivious and cache-aware algorithms," in *Cache-Oblivious and Cache-Aware Algorithms*, Dagstuhl, Germany, 2005, Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[73] M. Alvanos, G. Tzenakis, D. S. Nikolopoulos, and A. Bilas, "Task-based parallel h.264 video encoding for explicit communication architectures," in *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, 2011, pp. 217–224.

[74] J. Keller, C. Kessler, K. König, and W. Heenes, "Hybrid parallel sort on the cell processor," in *9th workshop on parallel systems and algorithms–workshop of the GI/ITG special interest groups PARS and PARVA*. Gesellschaft für Informatik e. V., 2008.

[75] N. Melot, C. Kessler, K. Avdic, P. Cichowski, and J. Keller, "Engineering parallel sorting for the intel scc," *Procedia Computer Science*, vol. 9, pp. 1890 – 1899, 2012, Proceedings of the International Conference on Computational Science, ICCS 2012.

[76] I. Walulya, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas, "Concurrent data structures in architectures with limited shared memory support," in *Euro-Par 2014: Parallel Processing Workshops*, Cham, 2014, pp. 189–200, Springer International Publishing.

[77] Y. Nikolakopoulos, M. Papatriantafilou, P. Brauer, M. Lundqvist, V. Gulisano, and P. Tsigas, "Highly concurrent stream synchronization in many-core embedded systems," in *Proceedings of the Third ACM International Workshop on Many-Core Embedded Systems*, New York, NY, USA, 2016, MES '16, p. 2–9, Association for Computing Machinery.

[78] S. Memeti, S. Pllana, A. Binotto, J. Kołodziej, and I. Brandic, "Using meta-heuristics and machine learning for software optimization of parallel computing systems: a systematic literature review," *Computing*, vol. 101, no. 8, pp. 893–936, 2019.

[79] O. Polychroniou, A. Raghavan, and K. A. Ross, "Rethinking SIMD vectorization for in-memory databases," in *Proc. of the 2015 ACM SIGMOD Int. Conf. on Management of Data*. 2015, ACM.

[80] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake, *GPU-Accelerated Database Systems: Survey and Open Challenges*, pp. 1–35, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[81] A. B. Maccabe, W. Zhu, J. Otto, and R. Riesen, "Experience in offloading protocol processing to a programmable NIC," in *Proceedings. IEEE International Conference on Cluster Computing*, 2002, pp. 67–74.

[82] A. Sapio, M. Canini, C. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. K. Ports, and P. Richtárik, "Scaling distributed machine learning with in-network aggregation," *CoRR*, vol. abs/1903.06701, 2019.

[83] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports, "Just say NO to paxos overhead: Replacing consensus with network ordering," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA, Nov. 2016, pp. 467–483, USENIX Association.

[84] A. V. Aho and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Commun. ACM*, vol. 18, no. 6, June 1975.

[85] H. Bos and K. Huang, "Towards software-based signature detection for intrusion prevention on the network card," in *Recent Advances in Intrusion Detection*, Alfonso Valdes and Diego Zamboni, Eds., Berlin, Heidelberg, 2006, pp. 102–123, Springer Berlin Heidelberg.

[86] I. Moraru and D. Andersen, "Exact pattern matching with feed-forward bloom filters," *J. Exp. Algorithmics*, vol. 17, Sept. 2012.

[87] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[88] I. Sourdis, V. Dimopoulos, D. Pnevmatikatos, and S. Vassiliadis, "Packet pre-filtering for network intrusion detection," in *2006 Symposium on Architecture For Networking And Communications Systems*, 2006, pp. 183–192.

[89] I. Sourdis, D. N. Pnevmatikatos, and S. Vassiliadis, "Scalable multigigabit pattern matching for packet inspection," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 2, pp. 156–166, 2008.

[90] G. Cormode, M. Garofalakis, P. J. Haas, C. Jermaine, et al., "Synopses for massive data: Samples, histograms, wavelets, sketches," *Foundations and Trends in Databases*, vol. 4, no. 1–3, pp. 1–294, 2011.

[91] G. Cormode, "Count-min sketch," in *Encyclopedia of Algorithms*, pp. 464–468. Springer, New York, NY, 2016.

[92] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, Berlin, Heidelberg, 2002, pp. 693–703, Springer-Verlag.

[93] T. Yang, J. Gong, H. Zhang, L. Zou, L. Shi, and X. Li, "Heavyguardian: Separate and guard hot items in data streams," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, New York, NY, USA, 2018, pp. 2584–2593, ACM.

[94] I. Sharfman, A. Schuster, and D. Keren, "A geometric approach to monitoring threshold functions over distributed data streams," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2006, pp. 301–312, ACM.

[95] M. Garofalakis, D. Keren, and V. Samoladas, "Sketch-based Geometric Monitoring of Distributed Stream Queries," *Proc. VLDB Endow.*, vol. 6, no. 10, pp. 937–948, Aug. 2013.

[96] N. Giatrakos, A. Deligiannakis, M. Garofalakis, I. Sharfman, and A. Schuster, "Prediction-based Geometric Monitoring over Distributed Data Streams," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2012, pp. 265–276, ACM.

[97] M. Garofalakis, "Approximate Geometric Query Tracking over Distributed Streams," *IEEE Data Eng. Bull.*, vol. 38, no. 3, pp. 103–112, 2015.

[98] T. Istomin, A. L. Murphy, G. P. Picco, and U. Raza, "Data Prediction + Synchronous Transmissions = Ultra-low Power Wireless Sensor Networks," in *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems*, 2016.

[99] Cisco, "Snort Rules and IDS Software Download," `https://www.snort.org/downloads/#rule-downloads`, 2018, Accessed: 2018-05-07.

Part II

Parallel Data Processing on Massively Parallel Servers

PAPER I

Charalampos Stylianopoulos, Magnus Almgren,
Olaf Landsiedel, Marina Papatriantafilou

Multiple Pattern Matching for Network Security Applications: Acceleration through Vectorization

Journal of Parallel and Distributed Computing (JPDC)
vol. 137, pp. 34 - 52, Elsevier 2020.

A preliminary version of this paper was published in:
the Proceedings of the 46th International Conference on Parallel Processing (ICPP)
Bristol, United Kingdom
August 14-17, 2017, pp. 472 - 482

2

Multiple Pattern Matching for Network Security Applications: Acceleration through Vectorization

Abstract

As both new network attacks emerge and network traffic increases in volume, the need to perform network traffic inspection at high rates is ever increasing. The core of many security applications that inspect network traffic (such as Network Intrusion Detection) is pattern matching. At the same time, pattern matching is a major performance bottleneck for those applications: indeed, it is shown to contribute to more than 70% of the total running time of Intrusion Detection Systems. Although numerous efficient approaches to this problem have been proposed on custom hardware, it is challenging for pattern matching algorithms to gain benefit from the advances in commodity hardware. This becomes even more relevant with the adoption of Network Function Virtualization, that moves network services, such as Network Intrusion Detection, to the cloud, where scaling on commodity hardware is key for performance.

In this paper, we tackle the problem of pattern matching and show how to leverage the architecture features found in commodity platforms. We present efficient algorithmic designs that achieve good cache locality and make use of modern vectorization techniques to utilize data parallelism within each core. We first identify properties of pattern matching that make it fit for vectorization and show how to use them in the algorithmic design. Second, we build on an earlier, cache-aware algorithmic design and show how we apply cache-locality combined with SIMD gather instructions to pattern matching. Third, we complement our algorithms with an analytical model that predicts their performance and that can be used to easily evaluate alternative designs. We evaluate our algorithmic design with open data sets of real-world network traffic: Our results on two different platforms, Haswell and Xeon-Phi, show a speedup of 1.8x and 3.6x, respectively, over Direct Filter Classification (DFC), a recently proposed algorithm by Choi et al. for pattern matching exploiting cache locality, and a speedup of more than 2.3x over Aho-Corasick, a widely used algorithm in today's Intrusion Detection Systems. Finally, we utilize highly parallel hardware platforms, evaluate the scalability of our algorithms and compare it to parallel implementations of DFC and Aho-Corasick, achieving processing throughput of up to 45Gbps and close to 2 times higher throughput than Aho-Corasick.

2.1 Introduction

Pattern matching is an essential building block for many security applications, such as antivirus programs or Network Intrusion Detection Systems (NIDS). In its core, pattern matching algorithms operate on two sets of input: (i) a predefined

set of patterns and (ii) an incoming stream of data and attempt to detect if any of the patterns exist in the stream. In this work, we focus on the problem of fixed-string, *multiple* pattern matching, i.e. the patterns are string literals and, differently from single pattern matching [1, 2], we are simultaneously tracking the presence of many patterns. In the context of Network Intrusion Detection Systems, the set of patterns are *signatures* of known malicious attacks (usually in the order of thousands) that the system aims to detect and the data stream is the reassembled stream of packets captured from the network interface.

Motivation and Challenges. Pattern matching represents a major performance bottleneck in many security mechanisms, especially when there is a need to employ analysis on the full packet's payload (Deep Packet Inspection). In intrusion detection, for example, more than 70% of the total running time in spent on pattern matching [3, 4]. Moreover, with the increasing interest in Network Function Virtualization (NFV) [5, 6], applications like firewalls and Network Intrusion Detection are now expected to be placed in the application layer of the control plane [7], where they need to rely on commodity hardware features for performance, like multi-core parallelism and vector processing pipelines.

Regarding the hardware features available in such commodity hardware, *vectorization* is gradually taking a more central role [8]. For example, architectures with SIMD instruction-sets now provide wider vector registers (256 bits with AVX) and introduce new instructions, such as gathers, that make vectorization applicable to a wider range of applications. Moreover, modern processor designs are shifting towards new architectures, like Intel's Xeon Phi [9], that, for example, supports 512 bit vector registers. On those platforms, vectorization is not just an option but a must, in order to achieve high performance [10]. In this work we introduce algorithmic designs to utilize these capabilities.

The introduction of *gathers* and other advanced SIMD instructions (cf. section 2.3) allows even applications with irregular data patterns to gain performance from data parallelism. For example, SIMD can speed up regular expression matching [11–13]. Here, the input is matched against a single regular expression at a time, represented by a finite state machine that can fit in L1 or L2 cache. Working close to the CPU is crucial for these approaches, otherwise the long latency of memory accesses would hide any computation speedup through vectorization.

The domain of multiple pattern matching for Network Intrusion Detection has challenging constraints that limit the effectiveness of these approaches: applications need to simultaneously evaluate thousands of patterns and traditional state-machine-based algorithms, such as Aho-Corasick [14], use big data structures that by far exceed the size of the cache of today's CPUs. The size of the patterns varies greatly (from 1-byte to several hundred byte patterns) and can

appear anywhere in the input. That is why SIMD techniques have not been previously considered for exact multiple pattern matching – with a few exceptions discussed in Section 2.8 – in the domain.

Moreover, the role of the use of memory hierarchies and of the mechanisms to access data more efficiently is a significant topic in computer science research and practice as well (c.f., e.g, [15, 16] and references therein). The results from [16] suggest that the memory hierarchy (caches, but also virtual address translation) have a significant effect on the actual running time of algorithms, even as simple ones as a random scan of an array. Inspired by those results, we also focus on the effects of the memory hierarchy and study how proper use of the latter can help matching algorithms perform much better in practice. In a similar spirit, it is known that the role of the patterns of data accesses for stateful processing and the impact of processing that avoids unnecessary move of data, also utilizing hardware for acceleration, is significant for efficiency (c.f., e.g, [17–19]) in general, and even more so in stream processing.

Approach and Contributions. In this paper, we introduce a vectorizable design of an exact pattern matching algorithm which nearly doubles the performance when compared to the state of the art, on SIMD-capable commodity hardware, such as Intel's Haswell processors or Xeon Phi [9]. Building upon recent work [20, 21] that take steps in addressing the cache-locality issues in pattern matching, we propose algorithmic designs for multiple pattern matching that bring together cache locality and modern SIMD instructions, to achieve significant speedups when compared to the state of the art. Combining cache locality and vectorization introduces new trade-offs on existing algorithms. Compared to traditional approaches that perform the minimum required number of instructions, but on data that is away from the processor, our approach, instead, performs more instructions, but these instructions find data close to the processor and can process them in parallel using vectorization.

Our work builds on a family of recent algorithms that take steps towards providing good cache locality for multiple exact pattern matching [20, 21]. They filter parts of the input streams using small, cache efficient data structures. We argue that, as a result, memory latencies are no longer the dominant bottleneck for this family of algorithms while their computational part becomes more significant. In this work, we follow a two-step approach. First, we propose a refined and extended method, which is able to benefit from vectorization while ensuring cache locality. Second, we design its vectorized version by utilizing SIMD hardware *gather* operations. To evaluate our approach, we apply our techniques to the DFC algorithm [20], as a representative example that outperforms existing techniques in Network Intrusion Detection applications, including [21], on which our proposed approach can be applied as well. We

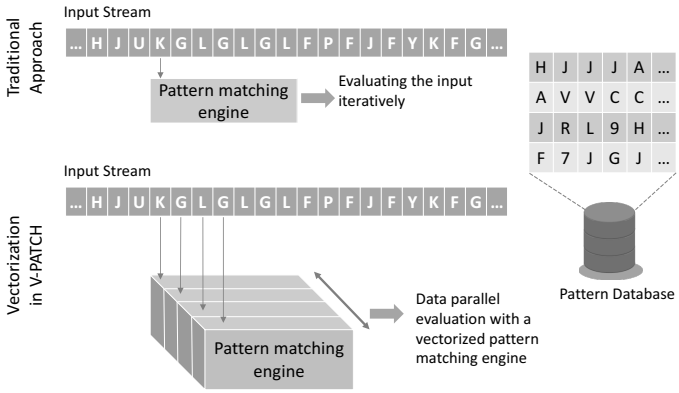


Figure 2.1: A general example of pattern matching at the top, and our proposed vectorized pattern matching approach at the bottom.

also include an analytical model that predicts the cost of both our scalar and vectorized algorithms, taking into account the number of malicious patterns given at startup. Finally, we deploy our algorithms on multi-core architectures and utilize all the available hardware parallelism, both within each core (with vectorization) and across many cores. A high-level illustration of our approach is shown in Figure 2.1.

In particular, we target the computational part of pattern matching for performance optimization and make the following contributions:

- We propose algorithmic designs for multiple pattern matching which (a) ensure cache locality and (b) utilize modern SIMD instructions.
- We devise a new pattern matching algorithm, based on these designs, that utilizes SIMD instructions to outperform the state of the art, while staying flexible with respect to pattern sizes.
- We introduce an analytical model to predict the performance of both our scalar and vectorized algorithms, based on the number of patterns. We evaluate the model with real-world data and find that it closely follows the observed trends.
- We implement the algorithm and thoroughly evaluate it under both real-world traces and synthetic data sets. We outperform the state of the art by up to 1.8x on commodity hardware and up to 3.6x on the Xeon-Phi platform.
- We evaluate the scalability of our algorithms when using all the parallelism offered by the platform and achieve up to 40 Gbps processing throughput on the Haswell platform and 45Gbps on the Xeon-Phi. We also design and evaluate

parallel implementations of existing algorithms (DFC and Aho-Corasick) and compare it against our algorithms. We find that our vectorized parallel version outperforms parallel Aho-Corasick by almost 2 times.

The remainder of the paper is organized as follows: Section 2.2 gives an overview of important pattern matching algorithms and background on vectorization. Section 2.3 describes our system model. In Section 2.4, we present our scalar approach leading to a new, vectorized design, described in Section 2.5. In Section 2.6 we introduce an analytical model to predict the performance of our scalar and vectorized algorithms. Section 2.7 presents our experimental evaluation on the performance of our algorithms under a variety of evaluations scenarios. In Section 2.8, we give an overview of other related work and we conclude in Section 2.9.

2.2 Background

In this section we present traditional approaches to pattern matching, followed by a brief description of the DFC algorithm (Choi et al. [20]) to which we apply our approach. Next, we introduce the required background on vectorization techniques.

2.2.1 Traditional approach to multiple-pattern matching

The most commonly used pattern matching algorithm for network-based intrusion detection is by Aho-Corasick [14]. It creates a finite-state automaton from the set of patterns and reads the input byte by byte to traverse the automaton and match multiple patterns. Even though it performs a small number of operations for every input byte, it implies– in practice and on commodity hardware – a low instruction throughput due to frequent memory accesses with poor cache locality [20]: As the number of patterns increases, the size of the state automaton increases exponentially and does not fit in the cache. In addition, the time to create such as state machine increases with the number of patterns [22] and quickly becomes a significant bottleneck. Nevertheless, the method is heavily used in practice; e.g., both Snort [23], one of the best known intrusion detection systems, as well as CloudFlare's web application firewall [24], use it for string matching.

2.2.2 Filtering approaches and cache locality in multiple pattern matching

Besides state-machine based approaches, there is a family of algorithms that rely on *filtering* to separate the innocuous input from the matches. Recent work focuses on alleviating the problem of long latency lookups on large data structures. Choi et al. [20] present a novel algorithmic design called DFC (Direct Filter Classification), that replaces the state machine approach of Aho-Corasick with a series of small, succinct summaries called *filters*. Such a filter is a bit-array that summarizes only a specific part of each pattern, e.g. its first two bytes, having one bit for every possible combination of two characters that can be found in the patterns. The algorithm is structured in two phases, the *filtering* and *verification*:

- In the *filtering* phase, a sliding window of two bytes over the input goes through an initial filter, as described above, to quickly evaluate whether the current position is a possible starting point of a match. The two-byte windows that passed the initial filter are fed to other, similar filters, each specializing on a family of patterns depending on their length. Since the filters are small (8KB each), they usually fit in L1 cache. Thus, the main part of the algorithm differs from Aho-Corasick and uses only cache-resident data structures, resulting in up to 3.8 times fewer cache misses [20].
- If a window of two characters passed all filters, there is a strong indication that it is a starting point of a match. For this reason, in the next *verification* phase, the DFC algorithm performs lookups on specially designed hash tables, containing the actual patterns and performs exact matching on the input and the pattern, to verify the match.

Other algorithms in this family, like [21] as well as this work, operate on the same idea: the input is filtered using cache resident data structures, and only the "interesting" parts of the input is forwarded for further evaluation.

2.2.3 Vectorization

Single Instruction Multiple Data (SIMD) is an execution model for data parallel applications, which utilizes processing units that operate on a vector of elements simultaneously, instead of separate elements at a time. SIMD instructions utilize the vector execution units, a separate pipeline found in modern processors that operates on multiple registers with almost the same cost as the equivalent scalar instructions. SIMD vectorization is a desirable goal in computationally intensive,

number-crunching applications, where computation is performed on independent data, *sequentially* stored in memory. However, until recently, most algorithms that did not follow this sequential access patterns were difficult to vectorize.

Vector instruction sets have evolved over time, introducing bigger registers and support for more complex instructions. Originally offering support for up to 128 bits, vector instruction sets are now extended to 256 bit-long vector registers and new generation platforms, such as the Xeon-Phi [9], support up to 512 bit-long vector registers, which indicates the vendor effort to accelerate applications that utilize data parallelism. Recently, vector instruction sets on commodity hardware have been enriched with the *gather* instruction [25] that enables accessing data from *non-contiguous memory locations* (described in detail in Section 2.3). Polychroniou et al. [26] study the effect of vectorization with the *gather* instruction on a series of data structures, such as Bloom-Filters, hash-table lookups, joins and selection scans, among others. We are building on these works with SIMD instructions and extend their design to pattern matching with the applications we focus on.

2.3 System model

In this section we introduce the assumptions and requirements that our approach makes on the hardware. We focus on mainstream CPUs, with vector processing units (VPUs) that support *gather* instructions. The latter make it possible to fetch memory from non-contiguous locations using only SIMD instructions[1].

The semantics of *gather* are as follows: let W be the vector length, which is the maximum number of elements that each vector register can hold. The parameters to the instruction are a vector register (I) that holds W indexes and an array pointer (A). As output, *gather* returns a vector register (O) with the W values of the array at the respective indexes. It is important to note that *gather* does not parallelize the memory accesses; the memory system can only serve a few requests at a time. Instead, its usefulness lies in the fact that it can be used to obtain values from non-contiguous memory locations using only SIMD code. This increases the flexibility of the SIMD model and allows to efficiency employ it for workloads previously not considered, i.e., where the memory access patterns are irregular. The alternative is to load the values using scalar

[1] In Intel processors, the *gather* instruction was introduced with the AVX2 instruction set and is included in the latest family of mainstream processors; *gather* also exists in other architectures, such as the Xeon Phi co-processor [9].

code, then transfer them one by one from the scalar registers into vector registers. Generally, switching between scalar and vector code is not efficient [26, 27].

Apart from *gather*, the rest of the instructions we use can be found across almost all the vector instruction sets available. Worth mentioning is the *shuffle* instruction, that makes it possible to permute individual elements within the vector register in any desired order. For example, we employ it for handling the input and output of the algorithm (cf. Section 2.5).

The size of the cache, especially the L1 and L2, is very important for the algorithmic design, as we describe later in Section 2.4. Common sizes in modern architectures is 32 KB of L1 data cache with 256 KB of L2 cache and we will use this as a running example. Our design is applicable to other cache sizes as well.

2.4 S-PATCH: a vectorizable version of the DFC algorithm

In this section, we begin by introducing S-PATCH (Scalar PATTern matCHing), an efficient algorithmic design for multiple pattern matching. It is designed with both cache locality and vectorizability in mind.

2.4.1 Overview

To enable efficient vectorization, we introduce significant modifications to the original DFC design. The key insight for the modifications, explained later in detail, is that small patterns will be found frequently in real traffic, so they should be identified quickly without adding too much overhead. On the other hand, long patterns are found less frequently, but detecting them takes longer and requires more characters from the input to pinpoint them accurately.

As in the original DFC, our approach has two parts, but it is organized as two separate rounds. In the **filtering** round, we examine the whole input and feed it through a series of filters that bear some similarities to DFC, but adapted to consider properties of realistic traffic, as motivated above. The **verification** round is as in DFC and performs exact matching on the full patterns that are stored in hash tables. Compared with DFC, S-PATCH focuses on efficient filtering in the first round, because this is the computationally intensive part of the algorithm that, as we show, can be efficiently vectorized. Splitting the two parts in separate rounds improves cache locality, since the data structures used in each round do not evict each other and, as shown in Section 2.5, makes vectorization more practical.

2.4.2 Filtering

In this first phase the goals are to (i) quickly eliminate the parts of the input that cannot generate a match and (ii) store the input positions where there is indication for a match. In general, key properties of the filtering phase include:

- Good filtering rate. A big fraction of the input is filtered out at this stage. This is important, in order to avoid performing verification frequently, as it has higher cost than filtering. The achieved filtering rate is directly dependant on the number of patterns inserted in each filter (see also the cost and hit rate predicted by the model described in Section 2.6).

- Low overhead. Every filter introduces additional computations and memory accesses, so there needs to be a balance between its overhead and the amount of input that is filtered out. Later in Section 2.6, our model quantifies the filtering overhead and the filtering rate, to help us maintain that balance.

- Size-efficiency. All the filters need to fit in L1 or L2 cache, while also leaving room for the input and the array for the intermediate results in cache. This is very important, because it ensures that the lookups on the filters will be fast and, as explained later, vectorization using the gather instruction will be feasible.

Our proposed filter design (cf. Figure 2.2) consists of three filters, each with a specific purpose. The first one stores information about the short patterns (less than 4 characters). It has one bit for every possible combination of two characters, and if a particular combination is the beginning of a pattern, the corresponding bit is set. Similarly, the second filter uses the same indexing and accounts for the longer patterns together with the third filter. An example of how filters are populated (in this example, Filter 2) is shown in Figure 2.3. In more detail on how we scan the input against the filters (cf. also Algorithm 2.1).

(A) First filter:

In the first part of the filtering, we examine two bytes of the input at a time and use them to calculate an index for filters 1 and 2. If the corresponding bit in the first filter is set, we directly store the current input position in an array for further processing (lines 5-7).

Filter 1 is responsible for patterns that are one to three bytes long and uses a two-byte index. For the case of one byte long patterns, we add in the filter all possible combinations of two byte pairs starting with that byte (e.g., if “A” is

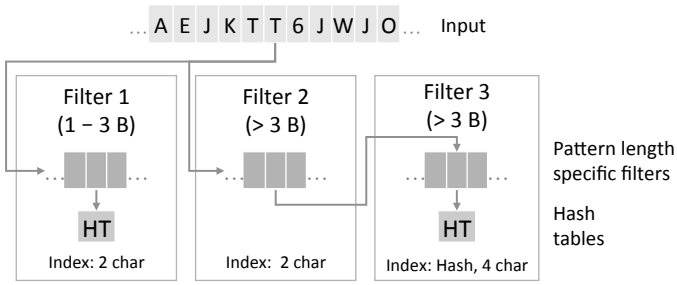


Figure 2.2: Filter Design of S-PATCH. HT stands for the *Hash Tables* that contain the full patterns.

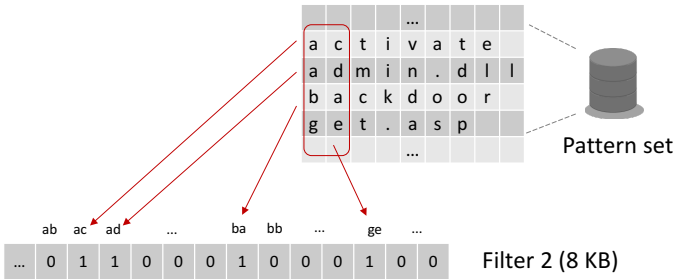


Figure 2.3: An example showing how Filter 2 is populated, for a given pattern set.

a pattern, we add “AA”, “AB”, “AC” etc. in the filter). For the case of three byte patterns, we use the first two bytes as input in the filter. During detection, if there is a hit in such a position, the verification phase that is explained later will check whether the third byte also matches.

(B) Second filter:

We also perform a lookup on the second filter using the same index, at line 8. A hit may indicate that we have a match with a longer pattern, but it may also be a false positive (e.g. compare the strings “**at**tribute” and “**at**tack”). Thus, before storing the current input position after a match with the second filter, the algorithm uses more bytes (in our case four) from the input stream with a third filter to gain stronger indications whether there is actually a match. Only when the match in the second filter is corroborated with a match from the third filter is the current position in the input stream stored for further processing (line 11).

Filters 1 and 2 both use two bytes as an index but are populated using patterns (one to three bytes for filter 1, the rest for filter 2). The reason is that Filter 2 is

Algorithm 2.1. Pseudocode for S-PATCH.

```
   Data: D: data to inspect
 1 # A_short : temporary array for short patterns
 2 # A_long : temporary array for long patterns
 3 for i=0, i <D.length, i++ do
 4 |  index = Read two bytes from pos i in D
 5 |  if (Filter1[index] is set) then
 6 |  |  Store i in A_short
 7 |  end
 8 |  if (Filter2[index] is set) then
 9 |  |  new_index = hash 4 bytes from input
10 |  |  if Filter3[new_index] is set) then
11 |  |  |  Store i in A_long
12 |  |  end
13 |  end
14 end
15 for i=0, i <A_short.length, i++ do
16 |  Verification for small patterns
17 end
18 for i=0, i <A_long.length, i++ do
19 |  Verification for big patterns
20 end
```

used a pre-filter for filter 3, to determine if it is necessary to perform the more expensive filtering that filter 3 requires.

(C) Choosing the index size:

Regarding filters 1 and 2, isolating the two bytes, is more costly than accessing 4 bytes, because it implies an extra shift or bit-masking to isolate them. However, the reason for choosing two bytes is to keep the size of the corresponding filter 8KB long so it can fit in cache. If we were to use either 3 or 4 bytes, that would leave us with two options: a) keep the same direct indexing, but the filter size would not fit in either L1 or L2 cache (2 MB filter size if using 3 bytes or 512MB if using 4 bytes). The cost of accessing even L3 cache is more than 10 times more than a L1 cache hit. b) not use direct mapping and use a hash function on a small filter that fits L1 cache (similar to filter 3, described next). In this case, we would have to pay the cost of computing the hash and we would likely end up with many collisions in the hash table.

Both of the cases mentioned above incur very high overhead, so we chose to prioritize keeping the filters in cache using two-byte, directly addressable filters. Fitting the filter in the L1 cache is also the reason why filters are compacted, and every element is one bit. The cost of extracting a specific bit is negligible if it allows us to keep the filter in L1 cache, especially for Filters 1 and 2 that are

accessed frequently. Similar arguments for the design of 2-byte indexed filters can be found in [20].

(D) Third filter:

The third filter is populated using the first four bytes of long patterns (four bytes or longer). For the third filter, the index is calculated differently; we cannot have a filter with all combinations of four bytes, due to cache-size limitations. Instead, we use a multiplicative hash function for the four bytes of input to compute the index in the filter, at line 9. As index, we use the hash value of those four bytes, modulo the number of bits in the filter. If, e.g., the third filter is also 8KB long, we end up with a two byte index that has been created as a hash of the first four bytes from each pattern. There is a trade-off between having a large enough filter to avoid collisions (thus providing a good filtering rate) and having it small enough to fit in cache. The reason why we choose four bytes as input will become clear in the next section (4 bytes fit in each one of the 32-bit vector register values).

Note that the performance of the filtering phase is intrinsically tied to the filter designs and the type of input. The reason why our proposed design is more effective is twofold. Short patterns, although few,[2] are likely to generate many matches. As an example, if strings like `GET` and `HTTP` are part of the pattern set, they will frequently be found in real network traffic. Treating them separately in a dedicated filter allows us to focus on the longer patterns in other filters. Long patterns, found more rarely, require more information to be distinguished from innocuous traffic.

2.4.3 Verification

After the filtering, all the possible match positions in the input have been stored in a temporary array. At this point, we need to compare the input at these positions with the actual patterns, before we can safely report a match. As mentioned before, the verification phase is as described by Choi et al. [20], except that it is now done in a separate round, after the current chunk of input has been processed by the filtering phase. For ease of reference we paraphrase here.

Among several optimizations, Choi et al. [20] use specially designed *compact hash tables* that are different for different pattern lengths. Translated to our improved filtering design, if the input at some position i passed the filtering, in

[2] 21% of Snort's v2.9.7 patterns are 1-4 bytes long [20].

the verification phase the algorithm will perform a match on the compact hash table that stores references to all the patterns of appropriate size. For example, if i passed the third filter that stores information on patterns that are four bytes or longer, in the verification phase, the algorithm performs a match on the compact hash table that stores patterns of four bytes or longer (lines 18-20). Each hash table is indexed with as many bytes as the shortest pattern that the hash table contains (in this case, four bytes of the input will be used as an index to the hash table). Each bucket in the hash table contains references to the full patterns and the algorithm has to compare each one of them individually with the input, before reporting a match. Eventually, the algorithm identifies all the occurrences of all the patterns, producing the same output as Aho-Corasick.

Example: As an example, if "abcdefgh" and "abcdklmn" are malicious patterns, the hash table at the index where "abcd" hashes to, will contain a pointer to a linked list that contains the two patterns that start from "abcd". The algorithm will then perform exact pattern matching between these patterns and the input. If necessary, more layers of filtering can be used in the hash tables themselves, as described in [20].

In general, the compact hash tables as we use them in this phase, do not fit L1 or L2 cache (but they might fit L3 cache) and accessing them incurs high latency misses. However, the success of the approach lies in the fact that the filtering phase will reject most of the input, so the algorithm resorts to verification only when it is needed (when there is a high probability for a match). That is why our efforts focus on the filtering part, where the data structures are close to the processor and can benefit from vectorization.

2.5 V-PATCH: Vectorized algorithmic design of the S-PATCH algorithm

In this section we outline the general design of V-PATCH and outline relevant optimizations. Next we describe how to parallelize the algorithms presented so far and discuss their runtime complexity.

2.5.1 General design

A basic issue when vectorizing S-PATCH is its non-contiguous memory accesses. The sequential version accesses the filters at nonadjacent locations for every window of two characters, whereas in a vectorized design W indexes are stored in a vector register (of length W), each pointing to a separate part of the data structure. For this reason, we use the SIMD *gather* instruction that allows us

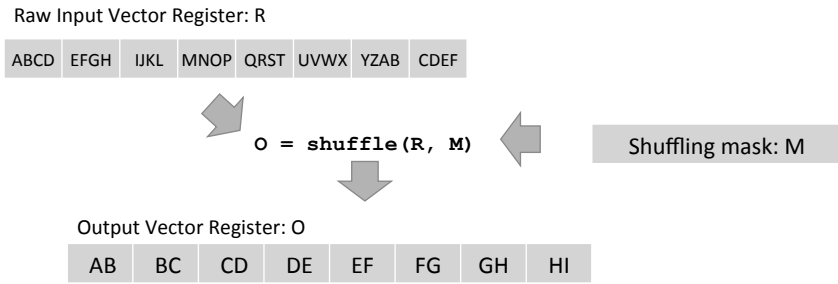


Figure 2.4: Input Transformation from consecutive characters to sliding windows of two characters.

to fetch values from W separate places in memory and pack them in a vector register. The gather instruction can operate efficiently if the data to be fetched can be found in cache [27], especially for architectures such as the Xeon Phi platforms (see description of platforms later in Section 2.7.1). In the absence of the gather instruction, the data can be fetched from memory using scalar instruction, but at the cost of mixing scalar and vector instructions [26].

Algorithm 2.2 gives a high level summary of the filtering phase of V-PATCH. The first step towards vectorizing the algorithm is loading the consecutive input characters from memory and storing them in the appropriate vector registers. Figure 2.4 shows the initial layout of the input and the desired transformation to W elements, each holding a sliding window of two characters. The transformation is efficiently achieved with the use of the *shuffle* instruction, allowing to manually reposition bytes in the vector registers (Algorithm 2.2, line 8). Note that we read overlapping segments from the input and produce W sliding windows from each segment, taking special care not to omit sliding windows that span across two uses of the *shuffle* instruction (in the example of Figure 2.4, the next use of *shuffle* will output IJ, JK, KL, etc.).

Once the vector registers are filled, the next step is to calculate the set of indexes for the filters. Note that every 2-byte input value maps to a specific *bit* in the filter, but the memory locations in the filter are addressable in *bytes*. A standard technique used in the literature [20, 28] is to perform a bit-wise right shift of the input value to the corresponding index in the filter. The remainder of the shift indicates which bit to choose from the ones returned. Having computed the indexes, we use them as arguments to the *gather* instruction that fetches the filter values at those locations (Algorithm 2.2, lines 9 and 13).

The *gather* instruction can take indexes of either 32 or 64 bits each. In the AVX2 instruction set, that supports up to 256 bit-long vector registers, using 32

Algorithm 2.2. Pseudocode for the V-PATCH filtering phase.

```
   Data: D: input data to inspect
1  # W : the vector register length
2  # A_short : temporary array for short patterns
3  # A_long : temporary array for long patterns
4  # M1⃗ : constant mask used to convert the input to 2 byte sliding window format
5  # M2⃗ : constant mask used to convert the input to 4 byte sliding window format
6  for i=0, i <D.length, i += W do
7  |  R⃗ = Fill register with raw input from D
8  |  Indexes⃗ = shuffle(R⃗,M1⃗)
9  |  V1⃗ = gather(filter1_address, Indexes⃗)
10 |  if at least one element in V1⃗ is set then
11 |  |  Store positions of matches in A_short
12 |  end
13 |  V2⃗ = gather(filter2_address, Indexes⃗)
14 |  if at least one element in V2⃗ is set then
15 |  |  NewIndexes⃗ = shuffle(R⃗,M2⃗)
16 |  |  Keys⃗ = hash(NewIndexes⃗)
17 |  |  V3⃗ = gather(filter3_address, Keys⃗)
18 |  |  if at least one element in V3⃗ is set then
19 |  |  |  Store positions of matches in A_long
20 |  |  end
21 |  end
22 end
```

bit indexes will fetch us 8 different memory locations while using 64 bit indexes will fetch 4 different locations. Since we want to have as much data parallelism as possible, we use up to 32 bit indexes in V-PATCH, which is also the reason why we use up to 4 bytes (32 bits) of input for filter 3.

As with the scalar algorithm, after a hit in the first or third filter we need to store the position of the input where a potential match occurred. We store the positions of the input that passed the filter from the set of W values in the register (lines 11 and 19). Here, we postpone the actual verification to avoid a potential costly mix of vectorized and scalar code, where the values from the vector registers need to be written to the stack and from there read into the scalar registers. Such a conversion can be costly and can negate any benefits we gain from vectorization [27].

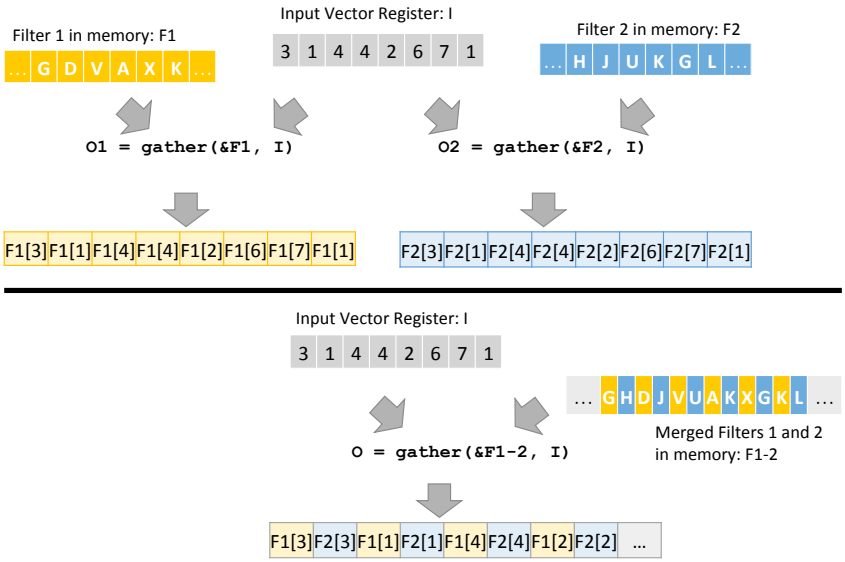


Figure 2.5: Figure describing the **filter merging** optimization. In the upper half, lookups on two filters require two gather invocations. Once the filters are merged in memory in the lower half, one gather brings information from both filters to the registers.

2.5.2 Design choices and optimizations

Regarding the number of *gather* instructions used, to optimize in latency, note that the first two filters (lines 9 and 13) are specifically designed to use the same indexes for a given input value in *gather* but different base addresses for the filters. Thus, with the **filter merging** optimization where the filters are interleaved in memory (at the same base address), we can merge lines 9 and 13 into a single *gather*, to bring the information from both filters from memory simultaneously. This optimization is not shown in the pseudo-code but depicted in Figure 2.5, giving an example in which a single *gather* instruction fetches information from both filters. Using bit-wise operations we can choose one filter or the other, once the data is in the vector register.

If at least one of the W values has passed the second filter, they need to be further processed through the third filter. Remember that the third filter uses a window of four input characters as an index. Thus, we load a sliding window of four input characters in each vector element in the register (line 15) and create the hash values that we use as indexes in the third filter (lines 16-17).

Not all of the values in the vector register are useful; only the ones that passed

the second filter need to be processed further by the third filter. This is a common challenge when vectorizing algorithms with conditional statements, since for different input we need to run different instructions. There are approaches [28] that manipulate the elements in the vector registers, so that they only operate on useful elements. For this particular algorithm, experiments with preliminary implementations showed that the cost of moving the elements in the registers out-weighted the benefits. Thus, we choose to speculatively perform the filtering on all the values and then mask out the ones that do not pass the second filter. In our evaluation (see later Section 2.7.3 Figure 2.8b), we observe that operating speculatively on all the elements is actually not a wasteful approach, especially with a large number of patterns to match.

Furthermore, to fully exploit the available instruction-level parallelism, we manually unroll the main loop of the algorithm by operating on two vectors (R_j) of W values instead of one, a technique that has proven to be efficient especially for SIMD code [28]. This has the benefit that, while the results of a *gather* on one set of W values are fetched from memory (line 9), the pipeline can execute computations on the other set of values in parallel.

2.5.3 Scaling across multiple threads

The description of V-PATCH so far focuses on how to utilize data parallelism within each core using vector instructions, but we can easily extended them to use multiple threads. With respect to that, we inherit the easily parallelizable property from DFC. DFC (as well as S-PATCH and V-PATCH) can start processing from any point in the input stream. Based on that, the algorithms presented in this section can be parallelized by splitting the received input into equal chunks and distributing it across the available threads. Then, each thread processes its own chunk independently. The only corner case is when malicious patterns spawn across two different chunks: to remedy this we allow each thread to continue processing each neighbouring thread's chunk, for as along as the longest pattern in the pattern set. Usually, the size of the longest pattern is very small (323 bytes in our evaluation), compared to the size of the each chunk (several MB). In Section 2.7.7 we show that our algorithms can scale with the number of threads.

2.5.4 Runtime complexity

Aho-Corasick, the standard algorithm in the literature, which we outlined in Section 2.2.1 processes every byte of input once and performs one state transition for each byte of input. Usually, the state of Aho-Corasick is stored in a 2-D array and the state transition can be done in constant time. As a result,

the runtime complexity of Aho-Corasick (when we only want to count the number of matches) is $O(n)$ where n is the size of the input to be scanned [22]. However, as described in Section 2.2.1, the drawback of Aho-Corasick is that the state cannot typically fit in cache, so in practice we expect a large constant in runtime complexity, because almost every memory access is a cache miss[3]. In fact, understanding and predicting the performance of such an algorithm on a real system with memory hierarchy is a challenging task and has given rise to models that incorporate the effect of caches or virtual address translation into the algorithm's complexity [16]. For example, [16] report the running time of random scan (which is similar to the random accesses in the state machine of the Aho-Corasick algorithmic) to follow a $O(n * logn)$ trend rather than a $O(n)$ trend as expected.

Filter-based algorithms, such as DFC, have worst case complexity $O(n * m)$ where m is the size of the longest pattern. This worst case complexity can happen when the longest pattern matches (or almost matches) at every position in the input, so there is a hit in the filters in every input byte. However, for typical cases of input text, the algorithm discards most of the input in the filtering phase, in linear time with a small constant because filters are more likely to be in memory. In fact, the whole design of such algorithms is focused around discarding most of the text early on so that the worst case complexity can be avoided. S-PATCH and V-PATCH have the same worst case complexity as DFC, but improve the performance in the average case even further (S-PATCH through better filtering and V-PATCH through vectorization). The performance in the average case is hard to predict and depends heavily on the number of patterns. For this reason, in the next section we introduce a performance model for S-PATCH and V-PATCH that gives us better insights in their expected performance.

2.6 Performance model

In order to better understand the runtime performance of the filter design we described above, in this section we introduce a simple model of the expected performance of the algorithm with respect to the number of patterns taken into account. We provide a model for both the scalar (S-PATCH) and the vectorized version (V-PATCH).

[3] As already mentioned another drawback of Aho-Corasick is the high runtime complexity of creating the state machine, which is much higher than creating the filters used by DFC and similar algorithms [20].

2.6.1 Usefulness

Our performance model is a useful tool to design and evaluate alternative filter architectures. As an example, for a given number of patterns, the model estimates the expected hit rate of the filters and the expected cost associated with filtering. Based on that, one can decide to add more filters in the design, or remove filters if their filtering ratio is low compared to the cost of accessing them. The model description that follows in this section refers to the filter design of S-PATCH and V-PATCH and assumes three filters, organized in the way shown in Figure 2.2. The same approach is applicable to other types of designs that use the same kind of filters as building blocks.

2.6.2 Filter hit rates

We start by estimating the hit rate of the filters, then use these rates to derive the overall performance model. We assume, for now, that both the input stream and the patterns are random. Then, if x is the number of patterns that are added to a filter, the probability that a bit in the filter is still zero is

$$p = (1 - \frac{1}{m})^x \tag{2.1}$$

where m is the size of the filter in bits. Equation 2.1 can be used for any filter size, but in the evaluation we use $m = 64K$ for all filters, including filter 3. This probability is derived by just considering the filter as a Bloom filter with a single hash function. In turn, the expected hit rate of a filter in the scalar case, i.e. the probability of accessing a single bit in the filter and finding it set to 1, is the complementary probability:

$$h(x) = 1 - p = 1 - (1 - \frac{1}{m})^x \tag{2.2}$$

Filter 1 in Figure 2.2 has a hit rate $h_1 = h(x_1)$ where x_1 is the number of patterns that are less than 4 bytes long. Note that, because filter 1 uses the first 2 bytes of the pattern as index, single-byte patterns need to be extended to 2 bytes. In order to do this, we create every possible combination of 2 byte characters starting with that single-byte pattern. For example, given the strings *BC* and *A*, we will set one bit at the index that corresponds to the position of *BC* and 256 bits on all indexes that start with *A* (*AA, AB, AC* etc.). As a result, x_1 accounts for all the patterns that are less than 4 bytes long and the number of extra patterns generated due to the presence of single-byte patterns.

Similarly, filter 2 in Figure 2.2 has a hit rate $h_2 = h(x_2)$, where x_2 is the number of patterns that are greater or equal to 4 bytes long. For filter 3, notice

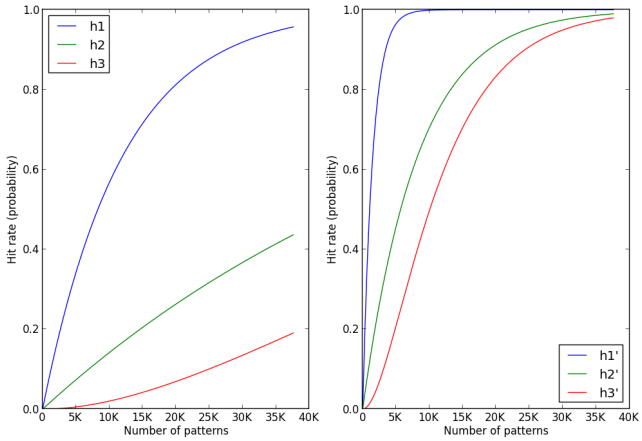


Figure 2.6: Expected hit rate for each filter in the scalar case (left) and the vectorized case (right).

that: (i) it has the same size and number of patterns as filter 2, (ii) accessing filter 3 requires a hit in filter 2 (see Figure 2.2) and (iii) it uses a different hash function from filter 2, so a hit in filter 2 tells nothing about the probability of a hit in filter 3. Based on that, the overall probability of having a hit in filter 3 is $h_3 = (h_2)^2$.

Turning to the vectorized case, remember that we have a hit in the filter if *at least one* of the W elements in the register hits the filter. Thus, the hit rate h' of a filter in the vectorized case is:

$$h' = 1 - (1 - h)^W \tag{2.3}$$

since $(1 - h)^W$ is the probability of having W consecutive misses.

Figure 2.6 shows the expected hit rates of the filters in the scalar and vectorized case for a varying number of random patterns. Here we assume that the size of each pattern is uniformly distributed between 1 and 50 bytes.

2.6.3 Overall cost

Knowing the hit rates of the filters allows us to model the overall per-byte cost of the algorithm. We model the filtering and the verification phases separately.

For each byte of input processed by S-PATCH, we identify the following main operations that need to be performed in the filtering phase: (i) compute the

indexes to filters 1 and 2 and access them, (ii) if there is a hit in filter 1, store the hit, (iii) if there is a hit in filter 2, compute the index for filter 3 and access it and (iv) if there is a hit in filter 3, store the hit. Those operations are the main factors in our model of the per-byte cost for the filtering phase of S-PATCH, which can be broken down as follows:

$$c_f = c_{1,2} + s_1 * h_1 + c_3 * h_2 + s_3 * h_3 \tag{2.4}$$

where $c_{1,2}$ and c_3 are the cost of computing the indexes and accessing for the first two ($c_{1,2}$) and the third filter (c_3) and s_1, s_3 are the cost of storing the indexes that produced a hit at filters 1 and 3, respectively. The cost of storing the hits is relatively small and we will exclude it from the model (but we will return to it in Section 2.7.4). Thus,

$$c_f = c_{1,2} + c_3 * h_2 \tag{2.5}$$

That leaves us with two constants that need to be computed, $c_{1,2}$ and c_3. We determine the value of these constants experimentally: we initialize use two sets of patterns (two different x values) and measure the cost of processing traffic data which were generated at random. As a result, we get with a system of two equations and two unknowns (the constants) which we solve to find the values for $c_{1,2}$ and c_3. These values are architecture dependent and are expected to differ between different architectures. Once we have determined them for a specific architecture, we use the same values to derive a single model that holds for different sets of input data and numbers of patterns, as we show later in Section 2.7.6. An alternative approach would be to derive the constants based on the expected number of instructions that each constant represents and the cost of each instruction on this architecture. However, this a complex and error-prone procedure, because we would need to factor in a lot of the micro-architecture details, e.g. the super-scalar pipeline and the out-of-order execution that the processor supports.

Similarly, the filtering cost for the vectorized case is

$$c'_f = c'_{1,2} + c'_3 * h'_2 \tag{2.6}$$

The cost of the verification phase is the same for both the scalar and the vectorized case. Remember that the algorithm reaches the verification phase when there is a hit on the first or the third filter. Verifying a hit involves a lookup in a hash table, the cost of which can be considered constant. Thus, the per-byte cost of verification can be modeled as follows:

$$c_v = c'_v = h_1 * V_{small} + h_3 * V_{large} \tag{2.7}$$

Table 2.1: Estimated values (in cycles) for the constants involved in the model, for the Haswell platform, c.f. Section 2.7.

	$c_{1,2}$	c_3	$c'_{1,2}$	c'_3	V_{small}	V_{large}
Estimated value (cycles)	3.8	26.0	3.1	4.3	7.7	110.7

where V_{small}, V_{large} are the cost of the hash table lookups for verification of small and large patterns, respectively. Again, we approximate these two constants experimentally using a system of two equations, similar to the way described above for the scalar algorithm.

In summary, the per-byte cost for S-PATCH is

$$c = c_f + c_v = c_{1,2} + c_3 * h_2 + h_1 * V_{small} + h_3 * V_{large} \tag{2.8}$$

and for V-PATCH:

$$c' = c'_f + c'_v = c'_{1,2} + c'_3 * h'_2 + h_1 * V_{small} + h_3 * V_{large} \tag{2.9}$$

The values we use for the constants are given in Table 2.1 (measured for the Haswell platform, c.f. Section 2.7). In Section 2.7 we evaluate the cost predicted by the model and show that it is accurate with respect to the one observed in practice.

2.7 Evaluation

In this section, we evaluate the benefits that our vectorization techniques bring to pattern matching algorithms. Our evaluation criteria are the processing throughput and the performance under varying number of patterns. We show the improvements of V-PATCH with both realistic and synthetic datasets, as well as with changing number of patterns. For a comprehensive evaluation, we compare the results from five different algorithms: the original Aho-Corasick ([14]; implementation directly taken from the Snort source code [23]), DFC (Choi et al. [20], summarized in Section 2.2.2), Vector-DFC (a direct vectorization of DFC done by us), S-PATCH (the scalar version of our algorithm, described in Section 2.4, that facilitates vectorization and addresses properties of realistic traffic that were not addressed before), and V-PATCH (the final vectorized algorithm described in Section 2.5).

2.7.1 Experimental setup

Systems: For the evaluation we use both Intel Haswell and Xeon-Phi. More specfically, the first system is an Intel Xeon E5-2695 (Haswell) CPU with 32KB

of L1 data cache, 256KB of L2 cache and 35MB of L3 cache. The platform has 14 cores on a single socket, with up to 2 threads per core, using hyperthreading. We use the ICC compiler (version 16.0.3) with -O3 optimization under the operating system CentOS. Unless otherwise noted, the experiments in this section are run on this platform. The second system is the Intel Xeon-Phi 3120 co-processor platform. Xeon-Phi has 57 simple, in-order cores at 1.1 GHz each, with 512-bit vector processing units. Each core supports up to 4 threads with hyperthreading. The memory subsystem includes a L1 data cache and a L2 cache (32KB and 512KB respectively) private to each core, as well as a 6GB GDDR5 memory, but no L3 cache. We compile with ICC -O3 (version 16.0.3) under embedded Linux 2.6. We are only using Xeon-Phi in native mode as a co-processor. The next versions of Xeon-Phi are standalone processors, so the problem of processor-to-co-processor communication is alleviated. In the following experiments, we first focus on the speedup achieved by a single hardware thread, through vectorization, then we discuss experiments with multiple threads.

Patterns: We use two sets of patterns: a smaller one, named *S1*, consisting of approximately $2,500$ patterns that comes with the standard distribution of Snort[4] [29] – the de-facto standard for network intrusion detection systems – and a larger one, named *S2*, with approximately $20,000$ patterns, that is distributed by `emergingthreats.net`. The patterns affect the performance of the algorithm and this is analyzed in detail in Section 2.7.3.

Data sets: In our evaluation, we use both real-world traces and synthetic data-sets. The real-world traces are the ICSX dataset [30, 31] (created to evaluate intrusion detection systems) and the DARPA intrusion detection dataset [32]. From ICSX, we randomly take 1GB of data from each of days 2 and 6 (thereafter named ICSX day 2 and ICSX day 6, respectively) and we also use 300MB of data from the DARPA 2000 capture. We are aware of the artifacts in the latter set, and the discussions in the community about its suitability for measuring the *detection capability* of intrusion detection systems [33]. In our experiments, we use it only for the purpose of comparing throughput between algorithms, allowing for future comparisons on a known dataset. The synthetic data set consists of 1GB of randomly generated characters.

An important point, considering the evaluation validity, is that, typically, not all the patterns are evaluated at the same time. In a Network Intrusion Detection System such as Snort, patterns are organized in groups, depending on the type of traffic they refer to. When traffic arrives in the system, the reassembled payload is matched only against patterns that are relevant (e.g. if the stream has HTTP

[4] We used version 2.9.7 for our experiments.

traffic, it is checked against HTTP related patterns, as well as more general patterns that do not refer to a specific protocol or service). To evaluate our algorithm in a realistic setting, we also pair traffic with relevant patterns. Since, in our datasets, most of the traffic is HTTP [30], we focus on HTTP traffic and match it against the patterns that are applicable based on the rule definitions. A similar approach can be used for other protocols (e.g. DNS, FTP), but we focus on HTTP traffic as it typically dominates the traffic mix and many attacks use HTTP as a vector of infection.

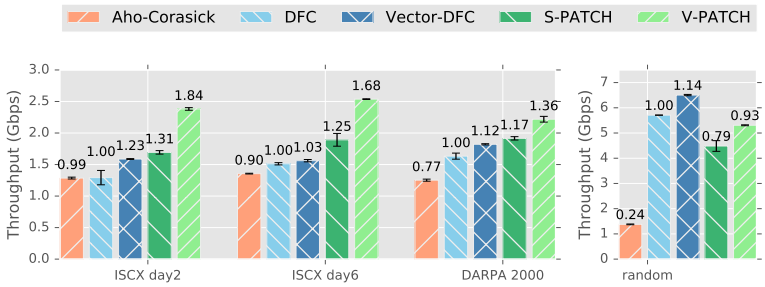
2.7.2 Overall throughput

In this section we compare the overall performance between the different algorithms. Using the HTTP-related patterns of each set gives us 2K patterns from pattern set *S1* and 9K patterns from pattern set *S2*. All algorithms count the number of matches. We use 10 independent runs of each experiment. We report the average throughput values, as well as standard deviation as error bars.

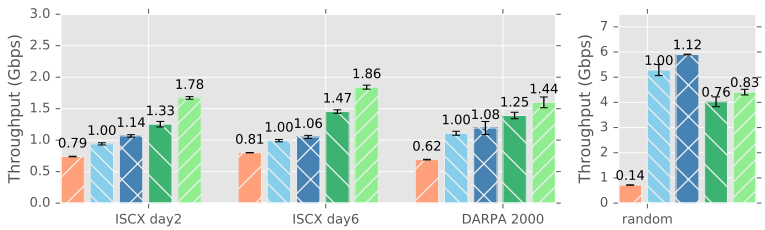
Figure 2.7a shows the throughput of all algorithms under realistic traffic traces and synthetic traces, when matched against the small pattern set (*S1*). In Figure 2.7b we use the bigger pattern set (*S2*). The numbers above the bars indicate the relative speedup compared to the original DFC algorithm.

We first discuss the results by only considering each pattern set and each traffic set separately. For realistic traffic traces, our vectorized implementation consistently outperforms the DFC algorithm by up to 1.86x (left parts of Figure 2.7), due to the parallelization we introduce in the filtering phase. The direct vectorization of the original DFC algorithm (Vector-DFC) has limited performance gain, because much of the running time of DFC is spent on verification and not filtering. This is the main motivation for introducing a modified version of DFC, in Section 2.4, focused on improving the filtering phase. By treating small, frequently occurring patterns separately and by examining more information in the case of long patterns, S-PATCH outperforms the original by up to 1.47x. More importantly, it allows for much greater vectorization potential, since the biggest portion of the algorithm's running time is shifted to efficient filtering of the input, and verification is done much more seldom.

Next, we evaluate the impact of the size of the ruleset on the overall throughput (comparing Figure 2.7a with Figure 2.7b). The overall throughput of the algorithms decreases, since the input is more likely to match and identifying every match consumes extra cycles. The performance of Aho-Corasick, in particular, decreases by more than 40%, because the extra patterns greatly increase the size of the state machine. The rest of the algorithms experience a 23-34% drop in performance.



(a) Snort web traffic patterns (2K).



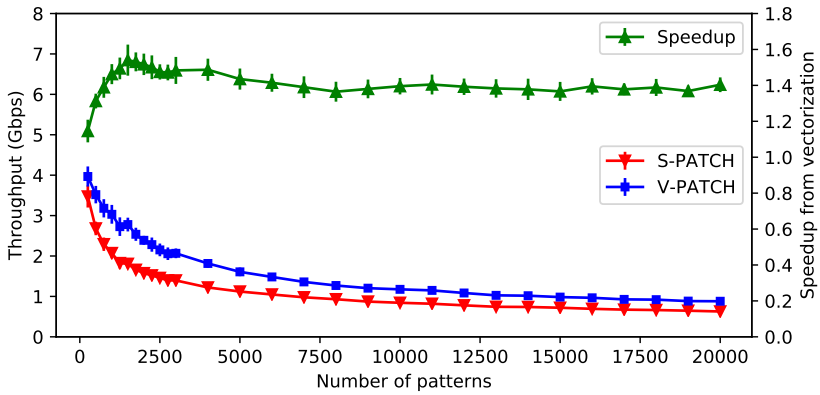
(b) ET open 2.9.0 web traffic patterns (9K).

Figure 2.7: Performance comparison between the different algorithms for public and random data sets, on the Xeon platform.

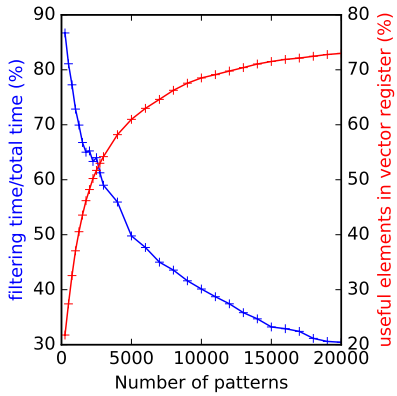
It is important to note that the performance gain of the algorithms (DFC versus Aho-Corasick, V-PATCH versus DFC) is influenced by the input as follows: when feeding the algorithms a data set that contains random strings, DFC significantly outperforms AC (right part of Figure 2.7). In this case, we do not expect to find many matches in the input and the filtering phase will quickly filter out up to 95% of the input. This is also the reason why the modified versions of the algorithm (S-PATCH and V-PATCH) perform less efficiently compared to what they do in the different input scenarios; the design of the two separate filters as described in Section 2.4 shows its benefits in more realistic traffic mixes. In turn, this poses interesting questions for the future in how to best design the filters based on the expected traffic mix. Still, the vectorized versions provides speedups over the scalar ones.

2.7.3 The effects of the number of patterns

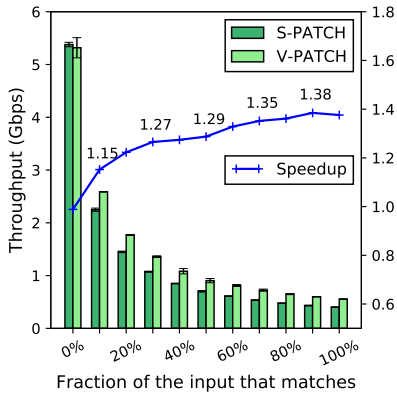
As shown in Section 2.7.2, it is important to account for the actual traffic mix the algorithms are expected to run upon when designing the filtering stage, as it



(a) Throughput as the number of patterns increases.



(b) Filtering to verification ratio and vectorization efficiency.



(c) Speedup from vectorization, as the numbers of matches in the input increases.

Figure 2.8: Figure a) compares the scalar and vectorized versions of our approach, as the number of patterns increases. Figure b) shows the filtering-to-verification ratio (left axis), as well as the average number of useful elements in the vector registers after filter 2 (right axis), as the number of patterns increases. Figure c) compares the scalar and vectorized approach, as the fraction of matches in the input increases.

has a large impact on the performance. As new threats emerge, more malicious patterns are introduced and the performance of the algorithm must adapt to that

change.

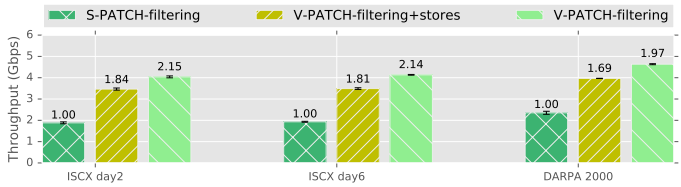
We measure the effects of the number of patterns on the two best performing algorithms and summarize the results in Figure 2.8a, also including the overall speedup of V-PATCH compared to S-PATCH. In this experiment, we randomly select the number of patterns from the complete set *S2* ($20,000$ patterns) in order to test our algorithms with as many patterns as possible. V-PATCH consistently performs better compared to S-PATCH, regardless of the number of patterns considered. Observe that:

- As the number of patterns increases, so does the input fraction that passes the filters. This causes the verification part, which is not vectorized, to take up more of the running time, essentially reducing the parallel portion and, by Amdahl's law [34], the benefit of vectorization. The portion of the running time spent in filtering, over the total running time is shown in Figure 2.8b (blue line).

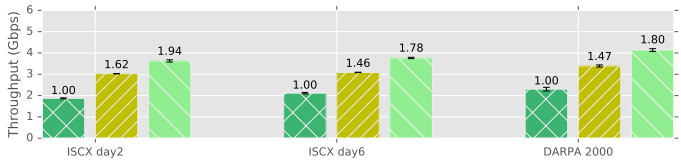
- As the number of patterns increases, the vectorization of the filtering becomes more efficient. Remember that V-PATCH will proceed with the third filter if at least one of the values in the vector register block passes the second filter. With a small number of patterns, we will seldom pass the second filter. When we do, it is likely we only have a single match, meaning that the rest of the values in the register are disabled and any computation performed for those values is wasteful work. Increasing the number of patterns results in more potential matches in the second filter and, as a consequence, less disabled values for the third filter and thus more useful work. In Figure 2.8b (red line) we measure this effect and show the average number of useful items inside the vector register every time we reach the third filter. Clearly, with an increasing number of patterns, the vectorization is performed mainly on useful data and therefore becomes more efficient.

- The two trends essentially cancel each other out, keeping the overall performance benefit of V-PATCH compared to S-PATCH constant after a point (Figure 2.8a), even though the optimized filtering gradually becomes a smaller part of the total running time. Eventually, the vector registers will always be full and we will not benefit from having more patterns. At this point the relative performance will stay constant. Our results indicate that this point is far beyond the number of patterns that current intrusion detection systems utilize.

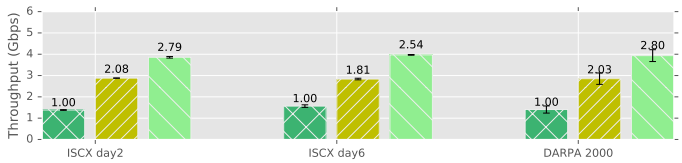
- A similar effect is observed when we keep the number of patterns constant, but increase the amount of matches in the dataset (Figure 2.8c). For this



(a) Snort web traffic patterns (2K).



(b) ET open 2.9.0 web traffic patterns (9K).



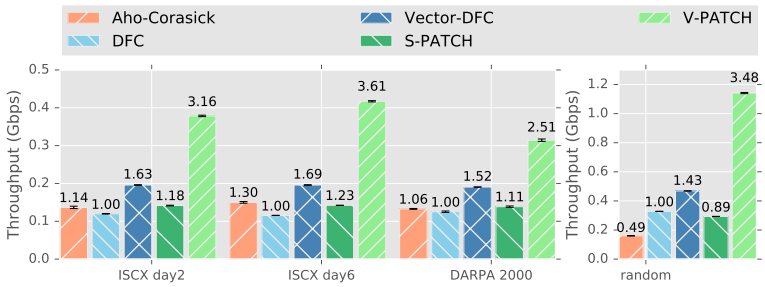
(c) Full pattern-set (20K).

Figure 2.9: Measuring the performance of the filtering part only. V-PATCH-filtering+stores includes the cost of storing the results of the filtering phase to temporary arrays.

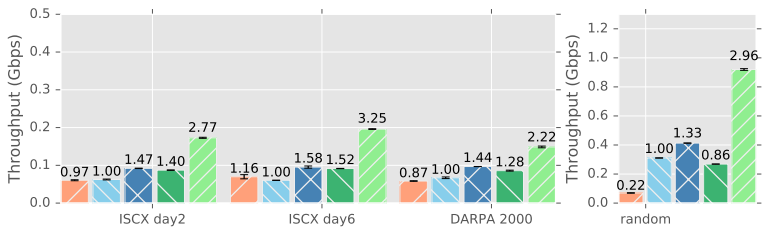
experiment, we created a synthetic input that contains increasingly more patterns, randomly selected from a ruleset of $2,000$ patterns. As more matching strings are inserted into the input, our vectorized portion of the algorithm becomes more efficient and the relative speedup compared to the scalar version slowly increases.

2.7.4 Filtering parallelism

In this section, in order to gain better insights about the benefits of vectorization, we measure the speedup gained in the filtering part in isolation. Figure 2.9 compares the filtering throughput of the scalar S-PATCH and V-PATCH, for pattern sets S1, S2, as well as the full pattern set (20K patterns). In the same figure, we also report the performance of the vectorized filtering, where we exclude the cost of storing the matches in the filtering phase in the temporary



(a) Snort web traffic patterns (2K).



(b) ET open 2.9.0 web traffic patterns (9K).

Figure 2.10: Performance comparison between the different algorithms for public and random data sets on the *Xeon-Phi platform*.

arrays. As we can see from the graph, the throughput of the filtering part is increased by up to a factor of 1.84x, on the small pattern set. Storing the matches of the filtering part in arrays comes with a cost; when it is removed, performance increases up to 2.15x for small pattern sets and up to 2.80x for the full pattern set. Even though there is a small decrease at the pattern set with 9K patterns (Figure 2.9b), the relative speedups of vectorized filtering increase with the number of patterns (Figure 2.9c).

2.7.5 Changing the vector length: results from Xeon-Phi

We have also evaluated the effectiveness of our approach on an architecture with a wider vector processing pipeline. The Xeon-Phi [9] co-processor from Intel supports vector instructions that operate on 512-bit registers, thus able to perform two times more operations in parallel, in the filtering phase.

Figure 2.10 summarizes the results from Xeon Phi, where the experiments are identical with those described in Section 2.7.2. Note that we report the throughput of a single Xeon-Phi thread. V-PATCH takes advantage of the wider

vector registers and outperforms the original scalar DFC algorithm, up to a factor of 3.6x on real data and 3.5x on synthetic random data.

As Xeon-Phi threads have much slower clock (1.1 GHz) and the pipeline is less sophisticated (e.g. there is no out-of-order execution), it is not surprising that the absolute throughput sustained by a single Phi thread is smaller than that of the single thread performance of the Xeon platform used in the previous experiments. When dealing with multiple streams in parallel, due to the higher degree of parallelism, the aggregated gain will naturally be higher, as indicated later in Section 2.7.7.

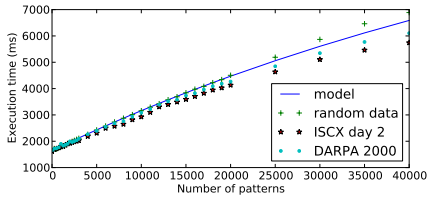
An interesting observation is that the DFC algorithm is sometimes slightly slower than AC on real data, where the number of matches in the input is significantly higher. In the original DFC algorithm, the filters are small and can easily fit L1 or L2 cache, and the hash tables containing the patterns are bigger, but still expected to fit L3 cache. In Xeon-Phi there is no L3 cache, so accesses to the hash tables in the verification phase are typically served by the device memory, negating the benefits of cache locality that is part of the main idea of the algorithm. Nonetheless, our *improved filtering* design reduces the number of times we resort to verification and access the device memory, thus resulting in 1.1x-1.5x increased throughput on realistic traffic, compared to the original DFC design.

2.7.6 Model evaluation

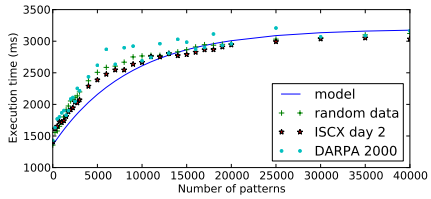
In this section, we evaluate the accuracy of our analytical model presented in Section 2.6. In the following experiments, we randomly generate up to 40K patterns and use different data sets, both real and synthetic. We show the normalized execution time for S-PATCH and V-PATCH, along with the cost predicted by the model.

Figures 2.11a and 2.11b show the cost of filtering for S-PATCH and V-PATCH, respectively. The figures show both the cost predicted by the model (given by Equations 2.5 and 2.6) as well as the cost measured using real and synthetic data. As predicted by the model, the cost of filtering for both versions is mostly affected by the hit rate of filter 2 (see also Figure 2.6). The cost of S-PATCH increases with the number of patterns, while the cost of V-PATCH flattens quickly (in this case, the hit rate of filter 2 is already close to 90% for more than 20K patterns and the vector registers are filed with mostly useful elements). Notice the different range in the vertical axis between S-PATCH and V-PATCH and the fact that, as the model predicts, the filtering part of V-PATCH is much faster than that of S-PATCH across any number of patterns.

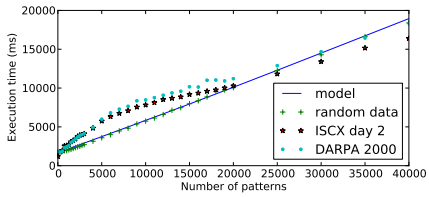
Similar to the above, Figures 2.11c and 2.11d show the total cost (in terms of



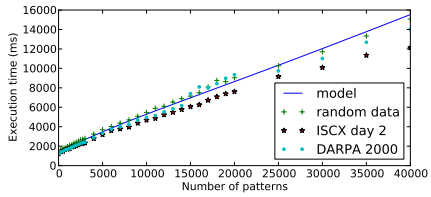
(a) S-PATCH filtering only.



(b) V-PATCH filtering only



(c) S-PATCH filtering and verification.



(d) V-PATCH filtering and verification

Figure 2.11: Real and predicted performance of S-PATCH and V-PATCH for different number of patterns.

execution time), including the cost of verification. The total cost for both follows an almost linear curve and is mostly dominated by the cost of verification, as predicted by the model (given by Equations 2.8 and 2.9). Since the model is fitted to random data, it predicts the cost of processing random data more closely compared to using realistic data (ISCX and DARPA data sets) where the traffic distribution is different. In this case of realistic data there is deviation from the model at around ten thousands patterns for the case of S-PATCH. Surprisingly, such deviation is not present for the case of V-PATCH. Also notice that, in most cases, processing real traffic is slightly faster than what is predicted by the model, most likely due to the different distribution of traffic.

Alternative filter designs: Having an accurate model to predict the overall performance of our algorithms allows us to easily evaluate different filtering architectures than the one we use for S-PATCH and V-PATCH (see Figure 2.2). We alter the model from Section 2.6 to predict a series of alternative designs, namely designs where we remove: (i) the filter for small patterns (Filter 1), (ii) one of the filters for long patterns (e.g. Filter 3) or (iii) all filtering whatsoever. By altering the model to cover these alternative designs, we can predict if, and at what number of patterns, it is beneficial to change our filtering design.

In Figure 2.12 we include the expected total execution time for 1GB of random data as predicted by the original model for S-PATCH, as well as the

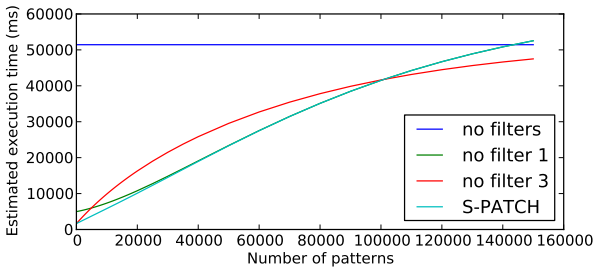
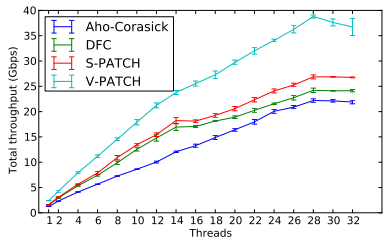


Figure 2.12: Prediction of the execution time of different filtering designs for S-PATCH, including designs where one or several of the filters are removed. Note the increased maximum number of patterns used in the horizontal axis.

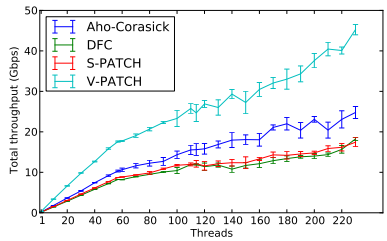
predictions for the alternative filtering designs discussed above. Note that we have extended the x-axis (number of patterns) to capture the trends at very large numbers of patterns, much larger than what is typically used in NIDS. Compared to our design (S-PATCH), removing Filter 1 has a small impact which is noticeable when less than twenty thousand patterns are used. Removing Filter 3 has initially a negative effect on performance, but the model predicts that it is a preferable choice when more than one hundred thousand patterns are used. This is reasonable since, when using so many patterns, filters are likely to be fully populated and have high hit-rates. In this case, the overhead of accessing the filter is not compensated by reducing the times we reach verification. If we remove all filters, we go to expensive verification for every input byte and the cost is prohibitively high, expect for the case of using more than one hundred and forty thousand patterns and all the filters are saturated. The trends also indicate that, for the number of patterns that are typically used in NIDS (one to ten thousand patterns) our original filtering design is a good choice, validating the design choices explained in Section 2.4. The respective alternative designs for V-PATCH follow trends similar to the ones in Figure 2.12.

2.7.7 Parallel execution

The experiments presented so far focus on the data parallelism achieved within a single thread, i.e. using vectorization and data parallelism within each core. In this section, we present experiments from a multi-threaded execution and demonstrate the scalability of our approach. We use the pthreads library in both of our shared memory architectures. As already mentioned in Section 2.5, we can easily parallelize DFC, S-PATCH and V-PATCH by splitting the available input in equal chunks. Nonetheless, it is important to evaluate the scalability of algorithms using multiple threads to show the effect of the underlying ar-



(a) Parallel execution on the Haswell platform.



(b) Parallel execution on the Xeon-Phi platform.

Figure 2.13: Parallel execution on the Haswell and Xeon-Phi platforms.

chitecture, e.g., resource sharing under hyper-threading. We also designed and evaluated a parallel version of Aho-Corasick, based on [22, 35].

For the following experiments, we used the ISCX day 2 data set and the S1 pattern set of 2K patterns. We split the input evenly across the available threads and report the total achieved throughput. We experiment on both the Haswell platform (14 cores, 28 threads) and the Xeon-Phi platform (57 cores, 228 threads). In all cases, our thread placement policy is to spread threads as much as possible, i.e. we first place each thread in each own core, then start placing up to two threads per core, etc.

Figures 2.13a and 2.13b show the results from the Haswell and the Xeon-Phi platforms respectively. In both platforms, all algorithms scale linearly while there is only one thread per core (up to 14 threads for Haswell and 57 threads for Xeon-Phi). After that, the scaling factor decreases, since threads that reside on the same core must share resources, such as parts of the execution units and the caches. For the case of the Haswell platform, we have also included tests where we spawn more software threads than the available hardware threads (over-subscription) and validate that we cannot get any more performance benefit. Nonetheless, all algorithms benefit from using the available thread-level parallelism in the system.

The relative speedup of our algorithms compared to Aho-Corasick follows roughly the speedup found in the single thread experiments, which also explains why Aho-Corasick is better than S-PATCH and DFC on the Xeon-Phi platform, based on the discussion in Section 2.7.5. Nonetheless, our scalar version (S-PATCH) is better than DFC in both platforms and our vectorized algorithm (V-PATCH) outperforms all other versions (up to 2 times better than Aho-Corasick), achieving up to 40 and 45 Gbps on the Haswell and Xeon-Phi platforms respectively.

2.8 Other related work

2.8.1 Pattern matching algorithms

Pattern matching has been an active field of research for many years and there are numerous proposed approaches. Aho-Corasick, explained before in Section 2.2.1 is one of the fundamental algorithms in the fields. There are variants of Aho-Corasick that decrease the size of the state transition table (for example [36]) by changing the way it is mapped in memory, but they come at an increased search cost, compared to the standard version of Aho-Corasick used in our evaluation. Other approaches apply heuristics that enable the algorithm to skip some of the input bytes without examining them at all, such as Wu-Manber [37] where a table is used to store information of how many bytes one can skip in the input. The main issue with these approaches is that they perform poorly with short patterns. For the problem domain investigated here, the patterns can be of any length and the algorithm must handle all of them gracefully. Moreover, in both Aho-Corasick and Wu-Manber algorithms, there is no data parallelism because there are dependencies between different iterations of the main loop over the input.

Recent algorithms [20,21] follow a different idea: Using small data structures that hold information from the patterns (directly addressable bitmaps in the case of [20], Bloom filters in the case of [21]), they quickly filter out the biggest parts of the input that will not match any patterns and fallback to expensive verification when there is an indication for a match. Our work is inspired by this family of algorithms, showing how they can be modified to perform better under realistic traffic and gain significant benefit from vectorization.

2.8.2 Regular expression matching

Apart from exact signature matching, intrusion detection systems also employ regular expression matching to detect attacks, because they offer more flexibility when describing the patterns. Regular expression matching usually utilizes finite automata, either deterministic (DFA) or non-deterministic (NFA). DFA's are fast, because every byte of input leads to only one state and their search complexity is O(n). However, the size of the state machine can grow exponentially with the number of regular expressions [38]. NFA's, on the other hand, construct a significantly smaller state in memory, but the search time is increased, because the state machine needs to evaluate several paths before finding a match. There has been significant work trying to find a compromise between search time

and memory use (for example [39]). Because regular expression matching is generally slow, Snort, a widely used NIDS, first applies exact pattern matching on the sub-strings that a regular expression contains, so most of the regular expressions do not have to be considered. The same approach is also followed in many proposed algorithms that target antivirus systems [40]. Thus, by improving the performance of exact pattern matching, we increase also the effectiveness of regular expression matching.

2.8.3 SIMD approaches to pattern matching

Even though pattern matching algorithms are characterized by random access patterns, SIMD approaches have been used before for pattern matching, especially in the field of regular expression matching. HyperScan [41] is a mature pattern matching framework that heavily relies on vector instructions for regular expression and fixed string pattern matching. Mytkowicz et al. [11] enumerate all the possible state transitions for a given byte of input to break data dependencies when traversing the DFA. Then they use the *shuffle* instruction to implement gathers and to compute the next set of states in the DFA. The algorithm is applied on the case where the input is matched against a single regular expression with a few hundreds of states and does not scale for the case of multiple pattern matching where we need to access thousands of states for every byte of input. Sitaridi et al. [12] use the same hardware gathers as we do, but apply them on database applications where the multiple, independent strings need to be matched against a single regular expression. There have been approaches that use other SIMD instructions for multiple exact pattern matching, but have constraints that make them impractical for the case of Network Intrusion Detection. Faro et al. [42] create fingerprints from patterns and hash them, but they require that the patterns are long, which is not always true for the typical set of patterns found e.g. in Snort.

The current paper is an extended version of [43] that introduces S-PATCH and V-PATCH. In this extended version, we also introduce and evaluate the analytical model (Section 2.6) and extend the current approaches with multi-thread parallelism (Section 2.7.7).

2.8.4 Other architectures

Outside the range of approaches that target commodity hardware, there is rich literature on network intrusion detections systems that are customised for specific hardware. For example, SIMD approaches that target DFA-based algorithms have been applied on the Cell processor [44], as well as FPGAs [45–47]. Most

notably, Graphics Processing Units (GPUs) are a popular target platform for pattern matching applications. GPUs are highly parallel architectures and are typically a good match for algorithms that are easily parallelizable, such as pattern matching. Lin et al. [48] present a parallelizable version of Aho-Corasick that removes the failure transitions (transitions taken in the state machine when a pattern is only partially matched). The algorithms begins the state-machine traversal at every input byte, in parallel. Bellekens et al. [49] compress the size of Aho-Corasick's state machine to reduce the communication cost between the CPU and the GPU. Aragon et al. [50] experiment with pattern matching on embedded GPUs that share the same physical memory as the CPU. Kouzinopoulos and Margaritis also experiment with pattern matching algorithms on GPUs and apply them on genome sequence analysis [46].

There is also significant work on GPUs that addresses pattern matching as part of a Network Intrusion Detection System. Vasiliadis et al. [45] build a GPU-based intrusion detection system that uses Aho-Corasick as the core pattern matching engine. Go et al. [51] use integrated GPUs and show that they are successful platforms for packet processing and Network Intrusion Detection. Jahmsed et al. [52] present Kargus, a custom NIDS that uses multiple GPUs and CPU cores. Papadogiannaki et al. [53] present a similar system and enhance it with a scheduler that dynamically decides the placement of packet processing tasks.

GPU parallelization has many similarities with vectorization; in fact GPUs offer more parallelism that can hide memory latencies. At the same time, it introduces additional challenges e.g. long latencies when transferring data between the host and the GPU. In this work we utilize vector pipelines that are already part of modern commodity architectures. Moreover, vectorization with CPUs requires careful algorithmic design that makes use of caches and advanced SIMD instructions. A main part of our work is showing how this problem can be tackled for the case of intrusion detection.

2.9 Conclusions

In this paper, we address the problem of multiple pattern matching and present an efficient, hardware-aware algorithm that utilizes the architectural features of commodity hardware to improve the processing throughput of Network Intrusion Detection Systems or other similar applications that employ pattern matching, e.g. antivirus systems. Specifically we introduce V-PATCH, a cache efficient filtering design, coupled with modern vectorization techniques that allow data parallelism within each processing core. We also provide an analytical model

for our algorithm that predicts the expected performance and can be used to create and evaluate new designs on-the-fly. The model also gives insights on the behaviour our algorithms that are difficult to capture without studying the effects of the number of patterns in the selectivity of the filters.

We thoroughly evaluate V-PATCH and its algorithmic design with both open data sets of real-world network traffic and synthetic ones in the context of network intrusion detection. Our results on Haswell and Xeon-Phi show a speedup of 1.8x and 3.6x, respectively compared to the state of the art and a speedup of more than 2.3x over Aho-Corasick, a widely used algorithm in today's Intrusion Detection Systems. Through the design and deployment of a series of multi-core parallel experiments, we also show that our approach can scale across many cores. Our vectorized version achieves up to 40 and 45 Gbps processing throughput on the Haswell and Xeon-Phi platforms, respectively and outperforms other parallel algorithms, including Aho-Corasick and DFC. Our experimental study provides fine-grained insights on different scenarios, including stress-tests under malicious traffic and thousands of malicious patterns. Finally, we show that our analytical model closely follows the experimental results and can thus be used as a valuable tool to create new filtering designs.

Acknowledgements

The research leading to these results has been partially supported by the Swedish Energy Agency under the program Energy, IT and Design, the Swedish Civil Contingencies Agency (MSB) through the projects RICS and RIOT, by the Swedish Foundation for Strategic Research (SSF) through the framework project FiC and the project LoWi, by the Swedish Research Council (VR) through the project ChaosNet, and from the European Community's Horizon 2020 Framework Programme under grant agreement 773717.

Bibliography

[1] Donald Knuth, James Morris, Jr., and Vaughan Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–350, 1977.

[2] Robert S. Boyer and J. Strother Moore, "A fast string searching algorithm," *Commun. ACM*, vol. 20, no. 10, pp. 762–772, Oct. 1977.

[3] Spyros Antonatos, Kostas G. Anagnostakis, and Evangelos P. Markatos, "Generating realistic workloads for network intrusion detection systems," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 1, pp. 207–215, Jan. 2004.

[4] João B. D. Cabrera, Jaykumar Gosar, Wenke Lee, and Raman K. Mehra, "On the statistical distribution of processing times in network intrusion detection," in *2004 43rd IEEE Conf. on Decision and Control (CDC)*, Dec 2004, vol. 1, pp. 75–80 Vol.1.

[5] Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Niels Bouten, Filip De Turck, and Raouf Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 236–262, 2015.

[6] Yong Li and Min Chen, "Software-defined network function virtualization: a survey," *IEEE Access*, vol. 3, pp. 2542–2553, 2015.

[7] James Kurose and Keith Ross, "Computer networks: A top down approach featuring the internet," *Pearson Addison Wesley*, 2016.

[8] "Intel vectorization tools," `https://software.intel.com/en-us/articles/intel-vectorization-tools`, 2015, Accessed: 2019-07-18.

[9] "Intel Xeon Phi product family," `http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html`, 2016, Accessed: 2019-07-18.

[10] "The importance of vectorization for Intel Many Integrated Core Architecture (Intel MIC architecture)," `https://software.intel.com/en-us/articles/the-importance-of-vectorization-for-intel-many-integrated-core-architecture-intel-mic`, 2013, Accessed: 2019-07-18.

[11] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte, "Data-parallel finite-state machines," in *Proc. of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2014, ASPLOS '14, pp. 529–542, ACM.

[12] Evangelia Sitaridi, Orestis Polychroniou, and Kenneth A. Ross, "SIMD-accelerated regular expression matching," in *Proc. of the 12th Int. Workshop on Data Management on New Hardware*. 2016, DaMoN '16, pp. 8:1–8:7, ACM.

[13] Peng Jiang and Gagan Agrawal, "Combining SIMD and Many/Multi-core parallelism for finite state machines with enumerative speculation," in *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York, NY, USA, 2017, PPoPP '17, pp. 179–191, ACM.

[14] Alfred V. Aho and Margaret J. Corasick, "Efficient string matching: An aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333–340, June 1975.

[15] Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran, "Cache-oblivious algorithms," *ACM Transactions on Algorithms (TALG)*, vol. 8, no. 1, pp. 4, 2012.

[16] Tomasz Jurkiewicz and Kurt Mehlhorn, "On a model of virtual address translation," *J. Exp. Algorithmics*, vol. 19, pp. 1.9:1–1.9:28, Jan. 2015.

[17] Gustavo Alonso, "How hardware evolution is driving software systems," in *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*, New York, NY, USA, 2019, DEBS '19, pp. 1–1, ACM.

[18] Tyler Akidau, "Open problems in stream processing: A call to action," in *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*, New York, NY, USA, 2019, DEBS '19, pp. 4–4, ACM.

[19] Hannaneh Najdataei, Yiannis Nikolakopoulos, Marina Papatriantafilou, Philippas Tsigas, and Vincenzo Gulisano, "Stretch: Scalable and elastic deterministic streaming analysis with virtual shared-nothing parallelism," in *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*, New York, NY, USA, 2019, DEBS '19, pp. 7–18, ACM.

[20] Byungkwon Choi, Jongwook Chae, Muhammad Jamshed, KyoungSoo Park, and Dongsu Han, "DFC: Accelerating string pattern matching for network applications," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, Santa Clara, CA, 2016, pp. 551–565, USENIX Association.

[21] Iulian Moraru and David G. Andersen, "Exact pattern matching with feed-forward bloom filters," *J. Exp. Algorithmics*, vol. 17, pp. 3.4:3.1–3.4:3.18, Sept. 2012.

[22] Charalampos S. Kouzinopoulos, Panagiotis D. Michailidis, and Konstantinos G. Margaritis, "Multiple string matching on a GPU using cudas," *Scalable Computing: Practice and Experience*, vol. 16, no. 2, 2015.

[23] "Snort rules and IDS software download," `https://www.snort.org/downloads`, 2016, Accessed: 2019-07-18.

[24] "Scaling CloudFlare's massive WAF," `https://www.scalescale.com/scaling-cloudflares-massive-waf/`, 2014, Accessed: 2019-07-18.

[25] "Gather Scatter operations," `http://insidehpc.com/2015/05/gather-scatter-operations/`, 2015, Accessed: 2019-07-18.

[26] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross, "Rethinking SIMD vectorization for in-memory databases," in *Proc. of the 2015 ACM SIGMOD Int. Conf. on Management of Data*. 2015, SIGMOD '15, pp. 1493–1508, ACM.

[27] Johannes Hofmann, Jan Treibig, Georg Hager, and Gerhard Wellein, "Comparing the performance of different x86 SIMD instruction sets for a medical imaging application on modern multi- and manycore chips," in *Proc. of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, New York, NY, USA, 2014, WPMVP '14, pp. 57–64, ACM.

[28] Orestis Polychroniou and Kenneth A. Ross, "Vectorized Bloom filters for advanced SIMD processors," in *Proc. of the Tenth Int. Workshop on Data Management on New Hardware*, New York, NY, USA, 2014, DaMoN '14, pp. 6:1–6:6, ACM.

[29] Martin Roesch, "Snort - lightweight intrusion detection for networks," in *Proc. of the 13th USENIX Conf. on System Administration*, Berkeley, CA, USA, 1999, LISA '99, pp. 229–238, USENIX Association.

[30] Ali Shiravi, Hadi Shiravi, Mahbod Tavallaee, and Ali A. Ghorbani, "Toward developing a systematic approach to generate benchmark datasets for intrusion detection," *Computers & Security*, vol. 31, no. 3, pp. 357 – 374, 2012.

[31] "UNB ISCX intrusion detection evaluation dataset," `https://www.unb.ca/cic/datasets/ids.html`, 2012, Accessed: 2019-07-18.

[32] "DARPA intrusion detection data sets," `https://www.ll.mit.edu/r-d/datasets/2000-darpa-intrusion-detection-scenario-specific-datasets`, 2012, Accessed: 2019-07-18.

[33] Matthew V Mahoney and Philip K Chan, "An analysis of the 1999 DARPA/Lincoln Laboratory evaluation data for network anomaly detection," in *Int. Workshop on Recent Advances in Intrusion Detection*. Springer, 2003, pp. 220–237.

[34] Gene M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proc. of the April 18-20, 1967, Spring Joint Computer Conference*, New York, NY, USA, 1967, AFIPS '67 (Spring), pp. 483–485, ACM.

[35] Charalampos Stylianopoulos, Linus Johansson, Oskar Olsson, and Magnus Almgren, "Clort: High throughput and low energy network intrusion detection on iot devices with embedded gpus," in *Secure IT Systems*, Nils Gruschka, Ed., Cham, 2018, pp. 187–202, Springer International Publishing.

[36] Marc Norton, "Optimizing pattern matching for intrusion detection," *Sourcefire, Inc., Columbia, MD*, 2004.

[37] Sun Wu and Udi Manber, "A fast algorithm for multi-pattern searching," Tech. Rep. TR-94-17, University of Arizona. Department of Computer Science, 1994.

[38] Gerard Berry and Ravi Sethi, "From regular expressions to deterministic automata," *Theoretical computer science*, vol. 48, pp. 117–126, 1986.

[39] Randy Smith, Cristian Estan, Somesh Jha, and Shijin Kong, "Deflating the big bang: fast and scalable deep packet inspection with extended finite automata," in *ACM SIGCOMM Computer Communication Review*. ACM, 2008, vol. 38, pp. 207–218.

[40] Sang K. Cha, Iulian Moraru, Jiyong Jang, John Truelove, David Brumley, and David G. Andersen, "SplitScreen: Enabling efficient, distributed malware detection," *Journal of Communications and Networks*, vol. 13, no. 2, pp. 187–200, Apr. 2011.

[41] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu, "Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA, 2019, pp. 631–648, USENIX Association.

[42] Simone Faro and M. Oğuzhan Külekci, *Fast Multiple String Matching Using Streaming SIMD Extensions Technology*, pp. 217–228, Springer, Berlin, Heidelberg, 2012.

[43] C. Stylianopoulos, M. Almgren, O. Landsiedel, and M. Papatriantafilou, "Multiple pattern matching for network security applications: Acceleration through vectorization," in *2017 46th International Conference on Parallel Processing (ICPP)*, Aug 2017, pp. 472–482.

[44] Daniele Paolo Scarpazza, Oreste Villa, and Fabriozo Petrini, "Peak-performance DFA-based string matching on the Cell processor," in *2007 IEEE International Parallel and Distributed Processing Symposium*, March 2007, pp. 1–8.

[45] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis, *Gnort: High Performance Network Intrusion Detection Using Graphics Processors*, pp. 116–134, Springer, Berlin, Heidelberg, 2008.

[46] Charalampos S Kouzinopoulos and Konstantinos G Margaritis, "String matching on a multicore GPU using CUDA," in *Informatics, PCI'09. 13th Panhellenic Con. on*. IEEE, 2009, pp. 14–18.

[47] Ioannis Sourdis and Dionisios Pnevmatikatos, "Pre-decoded CAMs for efficient and high-speed nids pattern matching," in *Field-Programmable Custom Computing Machines, FCCM 2004. 12th Annual IEEE Symposium on*. IEEE, 2004, pp. 258–267.

[48] Cheng-Hung Lin, Chen-Hsiung Liu, Lung-Sheng Chien, and Shih-Chieh Chang, "Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs," *IEEE Transactions on Computers*, vol. 62, no. 10, pp. 1906–1916, Oct 2013.

[49] Xavier JA Bellekens, Christos Tachtatzis, Robert C Atkinson, Craig Renfrew, and Tony Kirkham, "A highly-efficient memory-compression scheme for gpu-accelerated intrusion detection systems," in *Proceedings of the 7th International Conference on Security of Information and Networks*. ACM, 2014, p. 302, arXiv.

[50] Elena Aragon, Juan M. Jiménez, Arian Maghazeh, Jim Rasmusson, and Unmesh D. Bordoloi, "Pattern matching in opencl: Gpu vs cpu energy consumption on two mobile chipsets," in *Proceedings of the International Workshop on OpenCL 2013 & 2014*, New York, NY, USA, 2014, IWOCL '14, pp. 5:1–5:7, ACM.

[51] Younghwan Go, Muhammad Asim Jamshed, YoungGyoun Moon, Changho Hwang, and KyoungSoo Park, "APUNet: Revitalizing GPU as Packet Processing Accelerator," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, Boston, MA, 2017, pp. 83–96, USENIX Association.

[52] Muhammad Asim Jamshed, Jihyung Lee, Sangwoo Moon, Insu Yun, Deokjin Kim, Sungryoul Lee, Yung Yi, and KyoungSoo Park, "Kargus: A Highly-scalable Software-based Intrusion Detection System," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, New York, NY, USA, 2012, CCS '12, pp. 317–328, ACM.

[53] Eva Papadogiannaki, Lazaros Koromilas, Giorgos Vasiliadis, and Sotiris Ioannidis, "Efficient software packet processing on heterogeneous and asymmetric hardware architectures," *IEEE/ACM Transactions on Networking*, vol. 25, no. 3, pp. 1593–1606, June 2017.

PAPER II

Charalampos Stylianopoulos, Ivan Walulya, Magnus Almgren,
Olaf Landsiedel, Marina Papatriantafilou

Delegation sketch: a parallel design with support for fast and accurate concurrent operations

Adapted version of the paper that appeared in *the Proceedings of the 15th European Conference on Computer Systems (EuroSys)*, Article 4, pp. 1–16, ACM 2020.

3

Delegation Sketch: a Parallel Design with Support for Fast and Accurate Concurrent Operations

Abstract

Sketches are data structures designed to answer approximate queries by trading memory overhead with accuracy guarantees. More specifically, sketches efficiently summarize large, high-rate streams of data and quickly answer queries on these summaries. In order to support such high throughput rates in modern architectures, parallelization and support for fast queries play a central role, especially when monitoring unpredictable data that can change rapidly as, e.g., in network monitoring for large-scale denial-of-service attacks. However, most existing parallel sketch designs have focused either on high insertion rate or on high query rate, and fail to support cases when these operations are concurrent.

In this work we examine the trade-off between query and insertion efficiency and we propose *Delegation Sketch*, a parallelization design for sketch-based data structures to efficiently support concurrent insertions and queries. *Delegation Sketch* introduces a domain splitting scheme that uses multiple, parallel sketches to ensure all occurrences of a key fall into the same sketch. We complement the design by proposing synchronization mechanisms that facilitate delegation of insertion and queries among threads, enabling it to process streams at higher rates, even in the presence of high-rate concurrent queries. We thoroughly evaluate *Delegation Sketch* across multiple dimensions (accuracy, scalability, query rate and input skew) on two massively parallel platforms (including a NUMA architecture) using both synthetic and real data. We show that *Delegation Sketch* achieves from 2.5X to 4X higher throughput, depending on the rate of concurrent queries, than the best performing alternative and has up to 2.25X lower latency, while at the same time maintaining better accuracy at the same memory cost.

3.1 Introduction

To process high-rate, high-volume data it is often necessary (in terms of space and processing time) to perform analytics not on the data itself, but rather on a succinct representation thereof. For this purpose, sketches have been proposed as a way to maintain data streams' state and answer queries on it (e.g. frequency of elements in the input or top-k most common elements) using limited memory, at the cost of giving approximate, rather than exact answers.

A representative example that shows the usefulness of sketches is network traffic monitoring. As traffic flows into a big network at high rates, e.g., at the ingress router of a university network, an administrator [10] or some system, e.g., a Network Intrusion Detection System or an SDN controller that does dynamic

flow scheduling [33], might be interested to know at *any point in time* how many packets a given IP address has sent. Giving the exact answer to such a query requires storing all the incoming IP addresses and their counts, consuming memory proportional to the number of unique addresses. If, instead, an approximate answer is acceptable, a sketch can provide one with configurable error guarantees, using only a fixed amount of memory, without storing the IP addresses.

The literature on sketch-based algorithms offers a variety of ingenious techniques that mostly focus on the trade-off between memory consumption and accuracy [3, 5, 42]. Orthogonal to the need for small and accurate sketches is the need to process data at high rates. Thus, large research efforts focus on accelerating operations on the sketch, e.g., by using filters that process frequently found elements separately [32], which is important for many real-world input streams that are often highly skewed. As a result, high-throughput sketches are used in many applications, such as traffic monitoring [19, 24, 45] and data stream management tasks [7]. They are also used for communication reduction in distributed monitoring algorithms [13] and help with dimensionality reduction in machine learning algorithms [21].

Over the last few years, there has been a significant interest in parallel architectures to achieve sufficient high-speed processing. Multi-core platforms are adopted in many settings, from high-end servers [20] to low-end embedded devices [2] on the edge. Sketches can benefit from parallelism: e.g., regarding network traffic monitoring, state-of-the-art single-thread approaches [25, 32] achieve several millions of operations per seconds, which is enough to process traffic from 10Gbps links, but as link capacities increase to more than 100Gps, the need for multi-core processing becomes apparent. However, most of the work proposed on sketches focuses on the single-thread case and not on parallel settings. For existing parallel designs, we identify that there are conflicting requirements when considering both insertions and queries: parallel designs that perform efficiently when there are only insertions fail to scale when there are concurrent queries, while designs that favor queries cannot handle concurrent insertions efficiently. With the exception of very recent work [30] which we discuss in the related work section, most papers do not address concurrency between insertions and queries. This research gap is important, as many applications have need of both operations concurrently, including the IP-frequency-counting example above and other monitoring applications. In cases such as intrusion detection, applications must be able to handle high-rate traffic and support frequent queries, since the traffic characteristics might change abruptly and unpredictably [24].

In this paper, we identify and provide means to balance trade-offs involved in parallelizing sketches with respect to the number of threads, the rate of concurrent queries and the input distribution. We propose *Delegation Sketch*, a generic

parallelization scheme for sketches, that is able to process high input rates and scale efficiently even in the presence of high-rate concurrent queries, while at the same time achieving high accuracy with the same or lower memory requirements compared to existing schemes. *Delegation Sketch* can be applied on various sketches that support insertions and point queries [3,39,44,47] and aligns with the *regular* consistency specifications [22, 23, 29]. We make use of multiple parallel sketches and use hashing to ensure that the same key from different threads will end-up in the same sketch, allowing us to perform queries efficiently and more accurately. We also suggest a synchronization mechanism to delegate operations between threads, inspired by its uses in other concurrent data structure designs, e.g. in flat combining [17]. Our design: (i) allows threads to work on local data as much as possible, through the use of our proposed *Delegation Filters*, by aggregating multiple insertions on the same key locally without modifying any of the sketches; and (ii) combines multiple queries on the same key and serves them quickly. In particular, we make the following contributions:

- We study trade-offs in parallelizing sketches, with respect to concurrent insertions and queries and show the gap in existing designs. We demonstrate that the choice of parallelization does not affect only throughput and scalability, but also the accuracy of the result.

- We propose a generic parallelization design, *Delegation Sketch*, that scales with the number of threads, handles millions of insertions per second and is able to gracefully support concurrent queries.

- We provide a synchronization scheme that minimizes communication between *Delegation Sketch* threads and efficiently delegates operations on the sketch to other threads. We also leverage this synchronization mechanism to combine operations on the sketch to significantly improve performance and scalability.

- We provide an extensive experimental evaluation of *Delegation Sketch* and study it in connection to known parallelization designs on two massively parallel platforms with up to 72 and 288 threads, using both synthetic and real data. We show that *Delegation Sketch* supports up to 4X higher processing throughput and performs queries with up to 2.25X lower latency than the next best performing alternative. At the same time, *Delegation Sketch* has the same accuracy as the most accurate alternative, using the same amount of memory.

The rest of the paper is organized as follows: Section 3.2 gives the required background on sketches and describes the system model we target in this work. In Section 3.3 we analyze existing parallelization designs and motivate the need for *Delegation Sketch*, whose overview is given in Section 3.4. In Sections 3.5

and 3.6 we describe our design in detail. In Section 3.7 we present and analyze the results of our experimental evaluation. We discuss related work in Section 3.8 and conclude in Section 3.9.

3.2 Preliminaries

In this section, we describe the Count-Min sketch, a simple and efficient sketch, widely applicable in practice. We also describe a known extension to it, the Augmented Sketch, which includes techniques that we also adopt in our design. We finish this section by describing our system model.

3.2.1 The Count-Min and Augmented Sketch

The *Count-Min Sketch* [5] is a series of counters arranged in a 2-D array, with w columns and d rows. Every row is associated with one of d pairwise-independent hash functions $h_1, h_2, ..., h_d$, with h_i mapping keys from an input universe U to one of the w counters in row i. The sketch supports two operations: *insert*[1] and *point-query*. To insert a key K in the sketch, we increment the counter at position $h_i(K)$ at row i, for each one of the d rows. To perform a point-query on a key K, we hash the key with the same hash functions and select the counter at position $h_i(K)$ at row i, for each one of the d rows. The answer to the query is simply the minimum value among the selected counters, since that counter is closest to the true frequency of K, i.e. contains less "noise" from colliding keys. The answer to point-queries on any key K is always equal or higher than K's true frequency $f(K)$ and is lower than $f(K) + \frac{e}{w}N$ with probability $1 - \frac{1}{e^d}$, where N is the number of keys in the sketch [5]. Thus, one can configure the number of rows and columns to achieve error guarantees appropriate to the application.

In *Augmented Sketch* [32], Roy et al. couple a sketch with a filter to increase insertion throughput, especially when the input is highly skewed. The purpose of the filter is to efficiently keep track of a small number of keys that are frequently found in the input. When a new key needs to be inserted, if it is in the filter, then its frequency is updated there, without involving the sketch. Similarly, when performing a query on a key, if we find it in the filter then we report its frequency without querying the sketch for that key. Performing an operation on the filter is much faster than performing it on the underlying sketch, e.g. compared to the Count-Min Sketch that requires hashing a key multiple times.

[1] Aka *update* in the literature. We use the term *insert* throughout the paper.

3.2.2 System Model

Here we introduce the assumptions and requirements we make on the hardware platform, the application requirements and the consistency requirements.

Hardware Requirements: We assume a multi-core system with a finite set of threads $t_1, ..., t_T$ where T can be larger than the number of physical processors, along with a typical memory hierarchy, i.e. a L1 cache per thread, L2 and L3 caches shared between threads and main memory (either uniform on non-uniform). We consider an asynchronous shared memory system supported by a coherent caching model, through which a thread can access a shared variable not in the memory of the core where the thread is running. We also consider that no thread will arbitrarily fail or stop making progress.

We adopt the cache-register stream processing model [12], where input keys are continuously processed as they arrive and their frequency is continuously updated in the sketch. We assume that each thread has its own input sub-stream of keys. These sub-streams can originate from different sources or may have been extracted from a single stream in a previous part of the processing pipeline, either in software or in hardware, e.g., considering processing of packets coming from the network, many networks cards distribute the stream of packets to different CPUs [15, 16].

Application Requirements: *At any point in time, the application might query the frequency of a specific key in the total stream, i.e. across all sub-streams.* We assume that queries are much less frequent than the rate at which keys enter the system, but a query must be served even as new keys are being inserted and not at a later point in time when there are no more keys to insert. We also assume that each thread is serving one operation at a time: either an insertion of a new key from the stream, or a query.

Consistency Requirements: A query for the frequency of a key, performed by any thread, returns an approximation of the true frequency of the key, within the bounds provided by the underlying sketch. In the case of the Count-Min Sketch this includes the invariant that the answer is an over-approximation of the true frequency. The result must take into account all previous insertions of a key, across all sub-streams, but might or might not include insertions that overlap with the query, i.e., those that take place after the query has been issued and before it returns the result. This is a common assumption for concurrent data structures and it aligns with similar consistency specifications in literature, e.g. the *regularity* consistency specification [22, 23, 29]. In the case of sketches, the effects of not counting overlapping insertions are overshadowed by the fact that the answer is already an approximation of the true frequency.

3.3 Problem analysis

In this section, we summarize the existing parallelization designs that serve as baselines and we analyze their tradeoffs, in terms of the processing *throughput* of insertions and queries, the *accuracy* of the queries (i.e. the approximation error compared to the true frequency of a key) and the overall *scalability* of the design with the number of threads. We show that the existing designs have individual strengths but cannot efficiently handle the case of both insertions and queries, thus there is need for new parallelization designs.

3.3.1 Thread-local sketches

In the literature of sketch algorithms, most results focus on single thread performance and accuracy. When it comes to parallelization, most works [1, 32, 43] suggest the "*thread-local design*", where we have multiple sketches, one for each thread. Each thread inserts keys into its own sketch. To query a key, a thread queries every sketch and sums the results.

This design leads to very good scaling when there are only insertions, since each thread will work only on its own cache (sketches are usually small enough to fit L1 or L2 cache). However, the performance degrades significantly as soon as there are concurrent queries, since a querying thread needs to perform reads on all the sketches. The degradation worsens with the number of threads (since there are more sketches to read from) and becomes a major drawback in the highly parallel architectures we target in this work. Moreover, as shown in Section 3.5.1, this design leads to lower accuracy relative to its memory requirements, as each sub-query on a sketch introduces approximation errors that are then summed together.

3.3.2 Single-shared sketch

In work favoring queries over insertions [8, 37, 38] a single sketch is shared among all threads, a design henceforth referred to as the "*single-shared design*". Insert operations are slow, since threads require synchronization mechanisms, e.g. locks or (in the case of the Count-Min Sketch) atomic instructions and content on the memory of the sketch. Because of these reasons, in highly parallel environments targeted in this work, this design is not expected to scale with the number of threads when the input stream is inserted at high rates. However, queries are fast and accurate, since they can be answered immediately from the corresponding entries in the shared sketch and do not involve collecting results from multiple sketches.

Design name	Insertion Rate	Support for Queries	Scalability	Accuracy
Thread-local	high	low	high	low
Single-shared	low	high	low	high
Delegation Sketch	high	medium/high	high	high

Table 3.1: Comparison of parallelization designs.

3.3.3 The need for a new design

Based on the discussion above, it is evident that the existing parallelization designs focus on two extreme, opposing targets: they are effective either for applications that are only inserting keys at high rates with no queries (thread-local), or applications that will summarize a stream of keys once, and then only perform queries (single-shared).

In practice, many applications need to handle queries concurrently with insertions. Even though insertions are the most common operation for most applications (e.g., packet processing at high traffic rates), queries need to be handled concurrently as new keys are being inserted (e.g. IP counts must be queried at any time in traffic monitoring and flow scheduling). Moreover, support for high frequency queries, (e.g. one query every 1,000 insertions might mean one query every millisecond, depending on the input stream rate) is important for applications that need to react quickly to unpredictable changes [28, 36] or important events [24].

In order to serve such applications, we propose a new design, *Delegation Sketch*, that acts as a hybrid of the two designs mentioned earlier. We use multiple parallel sketches to allow our design to scale and perform insertions in parallel, but contrary to the thread-local design, a query needs to search for a key in only one of these sketches.

Table 3.1 summarizes the existing parallelization designs in comparison with *Delegation Sketch*. In the next section, we describe the main ideas and give an overview of our design.

3.4 Overview of *Delegation Sketch*

Our design bases on two techniques: Domain Splitting and Operation Delegation. We outline both here and detail in the subsequent sections.

3.4.1 Domain Splitting

To make queries faster, the number of sketches that a query has to search must be limited. To this end, we logically distribute the input domain of possible keys to

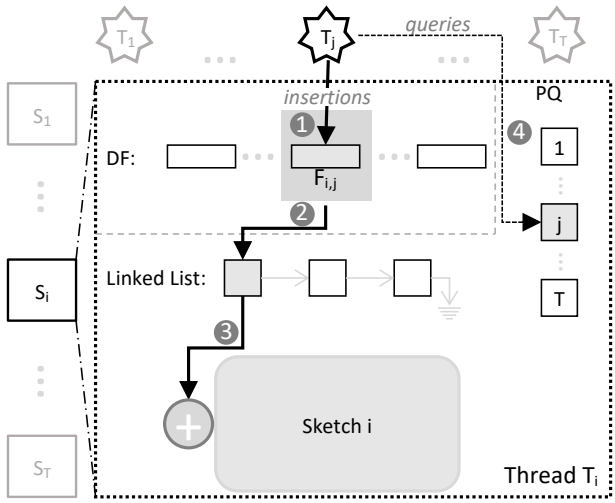


Figure 3.1: Outline of our design. DF stands for Delegation Filters and PQ stands for Pending Queries.

the T available sketches where each sketch is responsible for a set of keys. For every possible key K that can be found in the input stream of any thread t, we define as $Owner(K)$ the thread that is responsible for K. Finding the owner of a key can be as simple as $Owner(K) = K \ modulo \ T$. Every thread that wants to insert K will insert it into the sketch owned by $Owner(K)$. In this way, the same key (even if it is part of the input stream of different threads) will end up in a single sketch, making a query on that key a relatively cheap operation. In addition to making queries faster, splitting the domain of keys implies benefits on insertion speed, as well as on accuracy, for reasons we describe in Section 3.5.1.

3.4.2 Operation Delegation

As domain splitting requires a thread to insert into or query from an arbitrary sketch, we propose the use of filters (which we call *Delegation Filters*) that achieve this efficiently, minimizing inter-thread communication. An outline of our design, showing the series of Delegation Filters associated with Sketch i is shown in Figure 3.1. We give an overview of the purpose of filters here and explain the design and use of filters during insertion and query operations in detail in Section 3.6.

For every sketch, we keep a series of Delegation Filters, one for each thread.

The first purpose of Delegation Filters is to allow each thread to combine multiple insertions of the same key together, using only local updates without inter-thread communication. Instead of modifying the sketch every time there is an insertion operation on a key, threads aggregate the occurrence of the same keys in their stream (arrow 1 in Figure 3.1) and modify the sketch only when a sufficient number of keys have been aggregated. This is especially useful if the input is highly skewed: threads are doing insertions on the filters reserved for them most of the time, instead of modifying one of the sketches and causing contention.

The second purpose of Delegation Filters is to provide a unit of synchronization between a thread j that wants to insert keys to the sketch of thread i. The keys and their counts that thread j has aggregated in its filter will be inserted into a linked list of ready filters (arrow 2 in Figure 3.1) and eventually into the sketch by thread i (arrow 3).

Upon queries, a thread j will delegate a query operation on a key K and have it handled by another thread $i = Owner(K)$ (arrow 4). This design allows to optimize the number of times we have to search for the frequency of K in the sketch, by aggregating or "squashing" multiple pending queries on the same key to a single query operation on the sketch.

The use of delegation and the query "squashing" optimization are inspired by techniques used in concurrent data structures such as flat combining [17], where operations on a data structure are delegated for another thread that combines and performs them. Our design uses the Augmented Sketch (which we apply on top of the Count-Min sketch) as the underlying sketch, but different sketches that have the same interface (i.e. support insertions and point queries) can be used as well [3, 39, 44, 47]. In this work, we focus on point queries for frequency estimation, that are the basic type of queries supported by the Count-Min sketch. In the following section we detail the domain splitting technique and analyze its benefits, then describe the way we delegate operations.

3.5 Domain Splitting and benefits

The idea of splitting the domain of keys has been proposed for different scenarios and goals; e.g. Dobra et al. [9] apply it for join-size estimation and leverage approximate knowledge of the stream distribution, Thomas et al. [39] use it to handle architecture-specific constraints of the Cell processor. Here we utilize it in order to handle queries accurately and efficiently, as explained in the following subsections. The algorithmic implementation and the synchronization of the *Delegation Sketch* operations are described in Section 3.6.

3.5.1 Influence on the overestimation error

Here we study the accuracy of the different designs. We show that *Delegation Sketch* is: (i) more accurate than the fastest parallelization design (thread-local) (i) as accurate as the most accurate (albeit slower) parallelization design, while using the same amount of memory as those designs.

Due to the probabilistic nature of sketches, the result of querying for a key includes an amount of error. A query using the thread-local design involves querying all the sketches and summing the results. The intuition behind why domain splitting implies better accuracy than the thread-local design is that, by having all occurrences of the same key in a single sketch, it avoids aggregating the error from multiple sketches.

Reference sketch (single thread): Assume $f(i)$ is the frequency of key i, across the sub-streams of all threads. Based on [5], for a Count-Min sketch with w buckets and d rows, the estimate $\hat{f}(i)$ of key i is

$$f(i) \leq \hat{f}(i) \leq f(i) + \epsilon N \tag{3.1}$$

with probability $1 - \delta$, where $w = \frac{e}{\epsilon}$, $d = ln(1/\delta)$ and $N = \sum_{j \epsilon U} f(j)$ where U is the universe of keys.

Thread-local: This design uses T sketches of size $w * d$ each and the estimate when querying each sketch t is

$$f_t(i) \leq \hat{f}_t(i) \leq f_t(i) + \epsilon N_t \tag{3.2}$$

with probability $1 - \delta$, where f_t denotes the frequencies of keys that are in the sub-stream of thread t, $N_t = \sum_{j \epsilon U} f_t(j)$ and $\sum_{1 \leq T} f_t(i) = f(i)$. The total estimate $\hat{f}(i)$ is the sum of estimates from all the sketches so,

$$\sum_{1 \leq T} f_t(i) \leq \hat{f}(i) \leq \sum_{1 \leq T} f_t(i) + \epsilon \sum_{1 \leq T} N_t \tag{3.3}$$

or equivalently (by substitution)

$$f(i) \leq \hat{f}(i) \leq f(i) + \epsilon N \tag{3.4}$$

with probability at least $(1 - \delta)^T$.

This means that using the thread-local design (that uses T sketches with w buckets and d rows each) results to a similar bound as having one sketch with w buckets and d rows from Equation 3.1 and inserting all the elements in it.[2]

[2] In practice it is slightly better than that, because in the thread-local design we take the estimate (i.e. the minimum count) from each sketch and sum them, rather than summing the individual cells in a single sketch and then taking the estimate.

For that reason, the thread-local design is far from optimal, in terms of the ratio between accuracy and memory consumption it achieves.

Single-shared: Using the same total memory as in the thread-local design (by using a single sketch with d rows and $T * w$ buckets), for the single-shared sketch we have:

$$f(i) \leq \hat{f}(i) \leq f(i) + \frac{\epsilon}{T}N \tag{3.5}$$

with probability $1 - \delta$.

Domain splitting: In our design, by splitting the domain based on the number of threads, we have

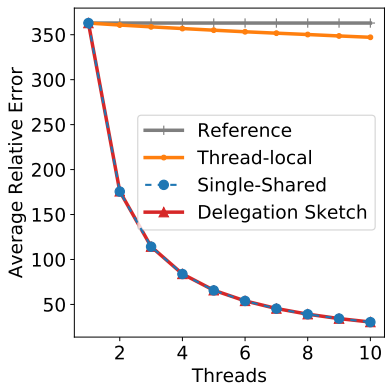
$$f(i) \leq \hat{f}(i) \leq f(i) + \epsilon N' \tag{3.6}$$

with probability $1 - \delta$, where N' is the total count of keys that hash to the same sketch as i. E.g. for a uniform distribution of keys, $N' = \frac{N}{T}$ and the bound is the same as in the single-shared design[3].

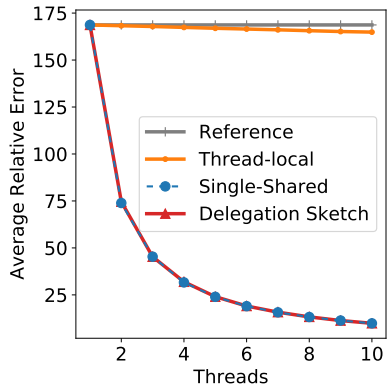
The aforementioned bounds for *Delegation Sketch* and thread-local design are in expectation and depend heavily on the input distribution. For this reason, we also examine the accuracy of those designs from an empirical point-of-view. In Figure 3.2 we show the difference in query error (in terms of the average relative error, also used in [32]) between the two approaches, using data from a uniform distribution (Figure 3.2a), as well as the Zipf distribution (Figure 3.2b). We have also included the "single-shared" sketch that uses a single sketch with the same total memory as the local-threads and domain-splitting designs, as well as the "reference" sketch that uses a single sketch with w buckets and d rows. For these experiments we used 600K keys taken from a universe of 100K distinct keys, then queried every key in that universe once. The memory footprint of each designs is shown in the table of Figure 3.2.

The results from Figure 3.2 align with the arguments above. The thread-local design has only slightly less error than the reference sketch, even though it uses T times more memory. Using domain splitting, the error decreases quickly based on the number of threads (equivalently, the number of sketches) in the system and its error is as low as that of a single-shared sketch that uses the same amount of memory.

[3]Later in Section 3.6 we introduce filters and their use in our design. We refine the bound of Equation 3.6 due to effects of filters in Section 3.6.3.



(a) Uniform distribution.



(b) Zipf distribution (skew=1).

Parallelization Design	Memory
Reference (single thread)	$w * d$
Thread-local	$w * d * T$
Single-shared	$w * d * T$
Domain splitting (Delegation Sketch)	$w * d * T$

(c) Memory consumption of the different parallelization designs we consider in the analysis. w and d are constant.

Figure 3.2: Average relative error as the number of threads increases. We also include the memory consumption for each design. The single-shared version has the same average relative error as the domain-splitting one.

3.5.2 Influence on query efficiency

Recall that in domain splitting, to perform a query on K, a thread will only have to query the sketch of $Owner(K)$, since all occurrences of K will have been inserted into that one (we explain how we perform query operations in detail in Section 3.6.2). For this reason, *domain splitting* leads to more efficient queries compared to the *thread-local* design where, as described earlier in Section 3.3, a querying thread will have to search for a key in multiple sketches. We support this claim experimentally in Section 3.7.4 where we present the latency of queries across different parallelization designs.

3.5.3 Influence on filter efficiency

Because with domain splitting the range of different keys that will be inserted in each sketch is smaller than U, the stream of keys that end-up on a sketch

appears more skewed, which increases the effectiveness of any filters that may be used by the underlying sketch (e.g. the Augmented Sketch), both in terms of throughput and accuracy. This effect on accuracy is studied in detail in the experimental evaluation, Section 3.7.2.

3.6 Operation Delegation and synchronization

The aforementioned benefits of domain splitting come with two challenges: (i) the fact that a thread will have to insert keys to another thread's sketch implies synchronization between threads, which needs to be done carefully in order to avoid bottlenecks and (ii) if the input is highly skewed, some keys will be much more common than others, which, in turn, implies that some threads' sketches will be more busy handling a large part of the input keys. In this section we describe how we use filters, which we call *Delegation Filters* to address both of these challenges.

Delegation Filters: For every sketch, we keep a series of Delegation Filters, one for each thread. We want searching for a key and incrementing its count to be as fast as possible, so we choose to implement them in a very simple manner: a filter is a pair of two arrays of fixed, small size. The first array holds the keys and the second one holds the count of that key, at the same index. By keeping the filters small we can search the whole filter for a key using only a few SIMD instructions, similar to [32].

We now explain in detail the way we use these filters, along with the description of the algorithmic implementations of the *Insert* and *Query* operations. We finish this section with a discussion on the memory consumption and overestimation error of *Delegation Sketch*.

3.6.1 Delegate insertions

For a thread j to perform the *Insert* operation on a key K, it first tries to insert it in the Delegation Filter $F_{i,j}$, reserved for thread j at the sketch owned by $i = Owner(K)$. To do this, it first searches Filter $F_{i,j}$ for key K. If it is found, it increments the count at that location, otherwise it adds K to an empty slot in the filter and sets the count there to one (lines 4-9 in Algorithm 3.1). If the filter is full, thread j adds a pointer to the filter in a multiple-producer single-consumer concurrent linked list maintained for filters that are ready to be inserted in the sketch of thread i (line 11). Thread j will then wait until the filter is *consumed* (i.e. until the keys in the filter and their respective counts have been flushed into the sketch by thread i). Note that, until the filter becomes full, i.e. the

Algorithm 3.1. Insert operation by thread j

```
1  Function Insert (key K) :
2  |  i ← Owner(K)
3  |  Filter_{i,j} ← Sketches[i].DelegationFilters[j]
4  |  // Filter_{i,j} is reserved exclusively for thread j
5  |  if K ∈ Filter_{i,j} then
6  |  |  Increment count of K
7  |  else
8  |  |  Add K in Filter_{i,j}
9  |  |  Set count of K to 1
10 |  end
11 |  if Filter.size = MAX_SIZE then
12 |  |  Sketches[i].LinkedList.push(pointer to Filter_{i,j})
13 |  |  while Filter.size = MAX_SIZE do
14 |  |  |  process_pending_inserts(j)
15 |  |  end
16 |  end
```

number of distinct keys in the filter is equal to the size of the filter, thread j can keep updating the filter without any communication with any other thread. This is because, every thread j has its own reserved filter associated with the sketch of thread i, thus alleviating the need for synchronization. The high-level pseudo-code of *Insert* is shown in Algorithm 3.1.

Periodically, threads check the linked list of full filters associated with their own sketch. This check can be performed at different points, e.g. after a certain timeout, after a successful completion of an insert or query operation, or while the thread is waiting for another thread to consume its filter (line 12 of Algorithm 3.1). E.g., thread j checks its own list of filters, in parallel while waiting for its filter to be consumed at line 14 of Algorithm 3.1. A high level pseudo-code of how threads process pending inserts is shown in Algorithm 3.2. Thread i traverses the list of pointers to filters that are ready to be inserted into its sketch. For every such filter, the thread iterates over the keys in the filter and adds their counts to the sketch (line 4-6 of Algorithm 3.2), using the semantics of the underlying sketch. Then, the thread removes any keys and their counts from the filter and marks it as empty (lines 7-8).

Claim 3.1. *All keys and their counts inserted in Delegation Filter j of thread i will be eventually inserted in the sketch of thread i, assuming threads continue to make progress.*

This is ensured by requiring thread i to have *exclusive access* on the filter while it is in the process of consuming the filter and inserting its contents in

Algorithm 3.2. Processing pending inserts by thread i

```
1 Function process_pending_inserts (thread i) :
2   while Sketches[i].LinkedList is not empty do
3     Filter ← Sketches[i].LinkedList.pop()
4     for each K in Filter do
5       Insert K to Sketches[i] (see Sec. 3.2)
6     end
7     Flush Filter
8     Filter.size ← 0
9   end
```

the sketch. We achieve this in the following ways: (i) for thread i to be able to consume a filter, it must first find it in its concurrent link-list of ready filters; thread j only adds it in the list when it is full, at which point it stops inserting items in it; and (ii) thread j will not start inserting keys in the filter unless it is marked as empty by thread i (line 8 of Algorithm 3.2).

3.6.2 Delegate queries

Similarly to *Insert*, for a thread j to query the frequency of key K, it first finds the thread $i = Owner(K)$. In order to accurately answer the query, the thread must count all occurrences of K, that can be found in: (i) the sketch owned by i and (ii) any of the T Delegation Filters associated with the sketch owned by i.

One option to achieve this is to have thread j search the sketch owned by i and its Delegation Filters. However, this would require synchronization between thread j and any of the T threads that might be concurrently accessing those Delegation Filters, as well as thread i that is inserting keys from the Delegation Filters into its sketch. Note that, allowing thread i to simply do this without any synchronization, might cause thread j to incorrectly "double count" occurrences of K: after thread j has counted X occurrences of K in a Delegation Filter, that filter might become full and get inserted into the sketch before thread j searches for K in the sketch, thus including X twice in the final answer.

Instead, we chose to delegate the query to thread i. Along with every sketch, we keep an array called *PendingQueries* of size T. Every item in the array holds a key, a counter (initially at zero) and a flag. Thread j adds key K at *PendingQueries[j]* and sets the flag there, to indicate that there is a pending query on key K (lines 4-6 of Algorithm 3.3). Thread j will then wait (checking its own list of filter and pending queries in the meantime at lines 8 and 9 of Algorithm 3.3) until the flag is set back to zero by thread i and read the answer to the query from the counter.

Algorithm 3.3. Query operation by thread j and processing of pending queries by thread p

```
1  Function Query (key K) :
2      i ← Owner(K)
3      PQ ← Sketches[i].PendingQueries
4      PQ[j].key ← K
5      PQ[j].count ← 0
6      PQ[j].flag ← 1
7      while PQ[j].flag = 1 do
8          process_pending_inserts(j)
9          process_pending_queries(j)
10     end
11     return PQ[j].count
12
13
14 Function process_pending_queries (thread p) :
15     PQ ← Sketches[p].PendingQueries
16     for t = 0; t < T; t + + do
17         if PQ[t].flag = 1 then
18             res ← 0
19             K ← PQ[t].key
20             for k = 0; k < T; k + + do
21                 F_{p,k} ← Sketches[p].DelegationFilters[k]
22                 res ← res + (count of K in F_{p,k})
23             end
24             res ← res + Sketches[p].get_estimate(K)
25             PQ[t].count ← res
26             PQ[t].flag ← 0
27         end
28     end
```

Threads periodically loop over their *PendingQueries* array and check if there is a pending query in each item of the array. For every pending query, threads get the key from the array, search all Delegation Filters (line 20-22) and the sketch for this key (using the semantics of the underlying sketch), report the result at the counter for that key and set the flag to 0. Note that searching T Delegation Filters and one sketch, even though it becomes a costly operation as the number of threads increases, is faster than searching T sketches, which is required in the thread-local parallelization design.

High level pseudo-code for *Query*, as well as the process of serving pending queries is shown in Algorithm 3.3.

Claim 3.2. *The query operation of* Delegation Sketch *takes into account all previous, non-overlapping insertions by any thread.*

This is because the query operation takes into account all possible locations where a key K can be, i.e., both the sketch of thread $Owner(K)$, and the Delegation Filters associated with that sketch. This includes Delegation Filters that are not yet full. In this case, the query operations might miss some overlapping insertions of K that are happening concurrently at a filter, but will include completed insertions. If the occurrence of a key has been inserted in the filter, all later queries will take that occurrence into account, either when reading it from the filter, or from the sketch if it has been moved there. Due to the domain splitting technique described in Section 3.5, the query operation does not need to search for the key in any of the other sketches or filters.

Claim 3.3. *The query operation of* Delegation Sketch *does not "double-count" the occurrences of any key.*

This is ensured by the fact that only one thread is responsible for searching for a key in the filters and the sketch. During this time no other thread can insert keys in the sketch, which would result in "double-counting".

Query optimization: query squashing

We now describe a simple optimization (not shown in Algorithm 3.3) that increases the performance of queries significantly, especially under conditions of high parallelism and input skew. When a thread i is done serving a delegated query on behalf of thread j, i.e. it has searched for key K in its sketch and the Delegation Filters associated with it, instead of just reporting the result to thread j, it iterates the array of pending queries to find other threads that have a pending query on the same key. Then, it reports the same result to those threads, without performing the actual search operations additional times, thus "squashing" the workload of multiple queries into one.

Note that this optimization does not report "stale" results for the point of view of the the thread that performs the actual query: the thread will only copy the same result to queries that are also pending, meaning that they are concurrent with the query of thread j. New queries that come after will trigger thread i to perform a new search of the sketch and the filters. However, the query squashing optimization might cause a query to not take into account a previous insertion, if that query gets "squashed" with another (overlapping) query that overlaps with the insertion. Let S_Q be the set of queries on the same key that overlap with query Q (including Q itself). Then, as an effect of the query squashing optimization, Claim 3.2 is adjusted to:

Claim 3.4. *The query operation Q of* Delegation Sketch *with the query squashing optimization takes into account all previous, non overlapping insertions with S_Q, by any thread.*

In practice, however, this effect has only a small impact on query accuracy, since it is limited to queries on the same key that overlap with each other, and is overshadowed by the accuracy loss that is intrinsic to sketches.

This optimization is made possible due to our design choice to delegate queries to other threads. In the next section, we evaluate its effects separately and show that it significantly increases the processing throughput, especially under highly skewed input.

3.6.3 Discussion on memory consumption and overestimation error

Since our delegation design needs memory for filters, and in order to make a fair comparison with other designs possible, we reduce the memory available to sketch accordingly, i.e., by using a smaller sketch, so that the total memory consumed is the same as the other designs we compare against. Similarly to [32], we achieve this by reducing the number of buckets at each row. Keeping the number of rows constant allows us to: (i) have the same δ probability bound for the estimate across all designs and (ii) keep the number of hashes used (hence the cost of insertions/queries on the underlying sketches) the same across all designs.

The use of filters, together with the fact that we adjust the size of the underlying sketch to make room for the filters, affects the existing error bound derived for domain splitting (Equation 3.6) in two opposing ways. We now quantify that effect and update the bound of Equation 3.6 here, following the analysis of [32] adapted to *Delegation Sketch*.

First, the underlying sketches are smaller, since we compensate for the memory of the filters by reducing the number of buckets at each row, as explained above. For a *Delegation Sketch* with w buckets, d rows, a filter size of f and T threads, we subtract $\lceil \frac{Tf}{d} \rceil$ buckets from the sketch. This increases the overestimation error of queries, but the effect is low for reasonably sized sketches and filters, as we show experimentally in Section 3.7.2.

Second, when querying the frequency of a key, some occurrences of the key will be found in the filters. The filters hold the exact number of occurrences and not an approximation, contrary to the query result found in the sketch for that key. That means that: (i) the overestimation error of a query decreases because parts of the answer will be given by the filters that hold exact counts and (ii) fewer

items will be inserted in the sketch, which reduces the approximation error due to collisions.

Overall, the error bound from Equation 3.6, due to the use of filter is now

$$f(i) \leq \hat{f}(i) \leq f(i) + \tilde{\epsilon}(N' - \tilde{N}) \tag{3.7}$$

where

$$\tilde{\epsilon} = \frac{e}{w - \frac{Tf}{d}} = \epsilon \frac{1}{1 - \frac{Tf}{wd}} > \epsilon \tag{3.8}$$

and N', ϵ are the same as in Equation 3.6. $\tilde{N}$ is the number of the items held in the filters of sketch i and depends on the skewness of the distribution and the size of the filters.

3.7 Evaluation

We present a detailed evaluation of the performance of *Delegation Sketch* with respect to accuracy and processing throughput. First, we describe our experimental setup, followed by the experimental results.

3.7.1 Experiment setup

Platform descriptions: We used two hardware platforms to evaluate our *Delegation Sketch*. *Platform A* is a dual socket NUMA server with 36 cores in total and 2-way hyper-threading at each core running at 2.1GHz, with 32KB L1 data cache, 256KB L2 cache and a 45MB shared L3 cache. It runs Ubuntu 16.04 and gcc v. 5.4. *Platform B* is a single socket, massively parallel Intel Xeon-Phi server with 72 cores, 4-way hyper-threading at each core running at 1.5GHz, using 32KB L1 data cache and 1MB L2 cache. It runs CentOS 7.4 and gcc v. 4.8.

Data sets: We used three sources of input data: *a) synthetic data* where the occurrence frequency of keys in the data set follow the Zipf distribution with a varying skew parameter. The Zipf distribution is widely used in the literature of sketch-based algorithms, as it captures the distribution of data related to many real world applications, such as packet counts, word count in a corpus of text, etc. *b)* two *real world data sets* taken from the CAIDA Anonymized Internet Traces 2018 Dataset [34]. From this trace, we use 22M packets that correspond to one minute of captured traffic from a high speed monitor. We extract the source IPs and source ports from the packet trace and use them as keys. This results in two input sets with very different characteristics: the frequencies at which IPs occur in the data set of IPs resemble a Zipf distribution with low skew,

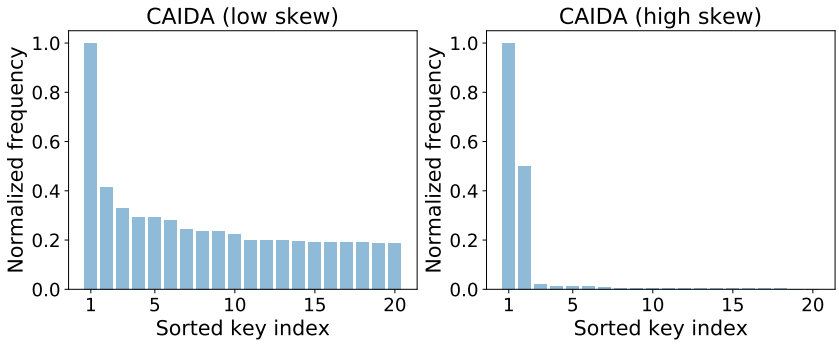


Figure 3.3: Normalized frequency of the 20 most frequent keys in the real world data sets used in the evaluation.

while the frequencies of ports in the data set of ports resemble a Zipf distribution with high skew. In Figure 3.3, we plot the normalized frequencies of the 20 most frequent keys for the two real world data sets.

As in [32], when we perform queries, we use the same distribution to determine on which keys we will perform them, i.e., we are more likely to perform queries for keys that are frequently found in the input stream.

Metrics: Our evaluation focuses on three metrics that are commonly used to characterize the performance of sketches: *accuracy*, *throughput* and *latency*. In Section 3.5, we already used the *average relative error* to evaluate the accuracy of different design choices. In this section, we additionally use the *absolute error per key* to indicate the over-approximation between the true frequency of a key in the stream and the frequency reported by the query. We report throughput as the number of operations (insertions or queries) per unit of time.

Parameters: In our experiments, we evaluate the effect of three main parameters: the *number of threads* in the system, the *skewness* of the input distribution and the *ratio of insertions vs queries* that each thread performs. Note that the *memory consumption* of each sketch is another important parameter that affects the performance of sketches in terms of accuracy and throughput. In order to have a fair comparison, we make sure that, for a given number of threads, *all versions use the same amount of memory. This includes all additional data structures involved, e.g., filters.* For *Delegation Sketch*, this is achieved as explained in Section 3.6.3. For the case of the single-shared sketch, we increase the number of buckets as we add more threads, in order to have the same total size in memory as the other designs that use multiple sketches.

Baselines: We study the performance of *Delegation Sketch* in connection to the single-shared and thread-local sketches, described in Section 3.3. As described above, we keep the total amount of memory constant between different

designs to ensure a fair comparison. We also include the Augmented Sketch using the thread-local design, i.e. we have one sketch and one filter per thread. In [32], the authors experiment with different filter sizes and evaluate the effectiveness of the filter. Based on that analysis, we use a filter size of 16 keys (and 16 counters) for all filters, including our Delegation Filters. In order to have a meaningful comparison with Augmented Sketch, we use Augmented sketch as the underlying sketch of *Delegation Sketch* i.e. every sketch in *Delegation Sketch* includes an additional 16 element filter. We also note that in our throughput evaluation (see later Section 3.7.3) we treat the Augmented Sketch baseline favourably: i.e., we do not attempt to enforce synchronization by making the filters thread-safe, i.e. the filters of Augmented Sketch can be accessed by any thread during queries. *Delegation Sketch* does not need special attention w.r.t. this, due to the synchronization mechanisms we describe in Section 3.6.

3.7.2 Comparing the accuracy of queries

In Section 3.5, we have already compared the accuracy of the different parallelization designs, in terms of Average Relative Error (ARE) and we have shown that, for the same total memory consumption, *Delegation Sketch* has very low ARE compared to the thread-local design. Moreover, *Delegation Sketch* is as accurate as the single-sketch design. We also showed that its accuracy increases with the number of threads.

Here we take a closer look at the accuracy of queries at each one of the input keys in our stream. For this experiment, we use a sketch with $d = 256$ and $w = 8$, use 4 threads and draw the input keys from the Zipf distribution with skew parameter 1. In Figure 3.4, we plot the error in the result of a query at every single key, using all the parallelization designs. For better presentation we have sorted the input keys based on their true frequency in descending order (e.g. the first 47K points in the x-axis correspond to the most frequent key, which has been seen 47K times in the input stream) and we plot the running mean of 1,000 keys.

Augmented Sketch and *Delegation Sketch* introduce no error on some of the most frequent keys in the stream, because of the filter used in the underlying sketch of both of those versions. Frequent keys are expected to be inserted in the filter and stay there most of the time. As a result, a query on those keys is more likely to report the true frequency of a key directly from the filter, rather than an approximation of it from the sketch. Note that this effect holds for more keys when using *Delegation Sketch* rather than Augmented Sketch. This is an effect of the domain splitting technique that reduces the range of keys that end-up at each sketch, thus making better use of the filter of the underlying sketch. *Delegation*

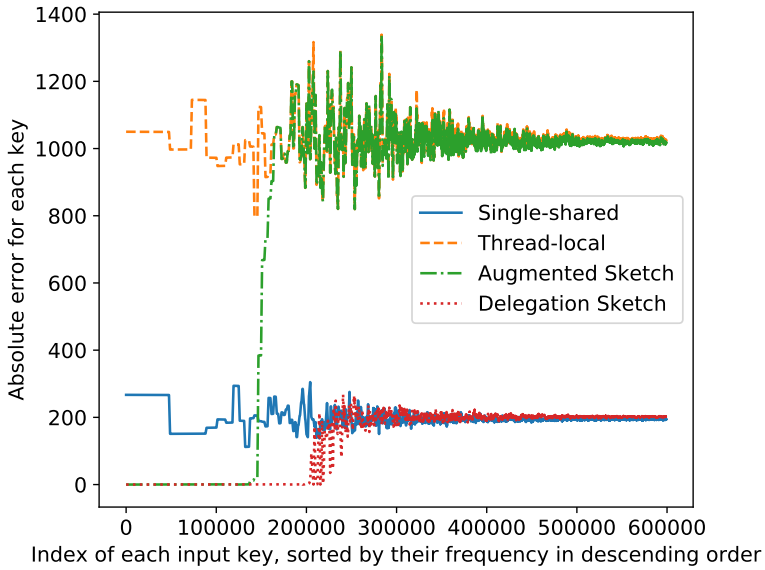


Figure 3.4: Error introduced for each key in the stream. The x-axis holds the indexes of each key in the stream, sorted by their frequency (descending order). The y-axis shows the absolute error added when performing a query on each key.

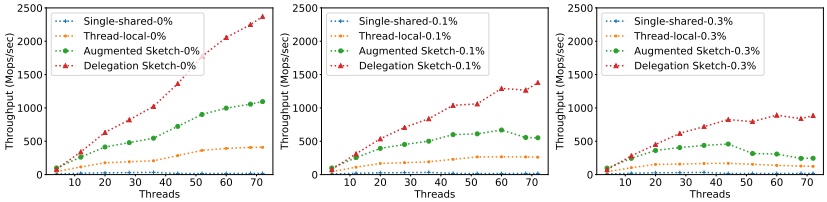
Sketch, as expected according to the argumentation in Section 3.5.1, continues to be one of the most accurate ones even for low frequency keys, despite the fact that it uses a smaller sketch to accommodate space for the Delegation Filters.

3.7.3 Processing throughput

Here we evaluate the throughput of *Delegation Sketch* and compare it with the baselines, across the three following dimensions: (i) scalability with the number of threads, (ii) query rate and (iii) input skew. Finally, we evaluate the effect of the Query Squashing method (Section 3.6).

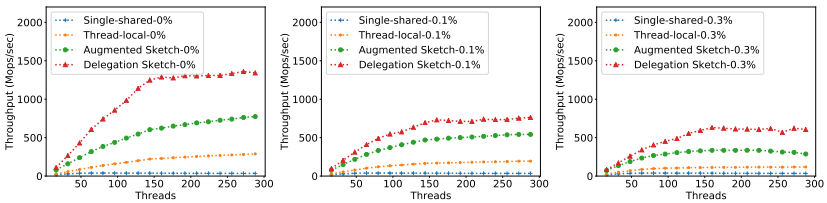
(A) Overall scalability

Figure 3.5 shows the overall scalability of the different baselines for Platform A. For this experiment, we use input keys coming from the Zipf distribution with skew parameter 1.5. We gradually increase the number of threads, as well as the ratio of queries vs insertions. We report the average number of operations per second, out of 10 runs. We omitted standard deviation because it was insignificant in most cases.



(a) 0% queries. (b) 0.1% queries. (c) 0.3% queries.

Figure 3.5: Platform A: Throughput and scalability comparison of all designs, using data from the Zipf distribution (skew=1.5).

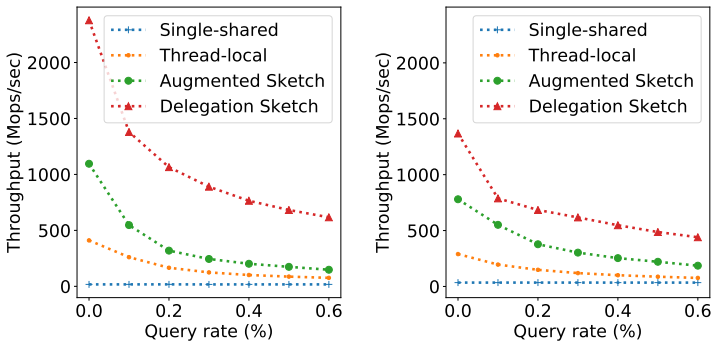


(a) 0% queries. (b) 0.1% queries. (c) 0.3% queries.

Figure 3.6: Platform B: Throughput and scalability comparison of all designs, using data from the Zipf distribution (skew=1.5).

In Figure 3.5a, we present the results from the execution of a workload that contains only insertions. We see that the single-shared parallelization design cannot scale with the number of threads, while the thread-local designs (including parallel Augmented Sketch), as well as *Delegation Sketch* benefit from paralellization. This is in accordance with the tradeoff analysis of Section 3.3. Even in the absence of queries, *Delegation Sketch* is up to 2X better than the next best baseline (Augmented Sketch), especially with more than 10 threads.

The introduction of even a small percentage of queries (Figures 3.5b and 3.5c) has a significant effect on processing throughput and scaling. With the exception of the single-shared design, the absolute throughput of all other designs is reduced. The thread-local design and the parallel Augmented Sketch stop scaling after approximately 40 threads in the case of the 0.3% query workload (Figure 3.5c) and actually perform worse with more threads. This is because increasing the number of threads introduces more sketches to search when serving a query. On the contrary, *Delegation Sketch* continues to benefit from parallelization, achieving up to 4 times higher throughput than the best performing baseline (Augmented Sketch). Also note that, on this platform, *Delegation Sketch* continuous to scale even under the effect of hyper-threading (that starts at



(a) Platform A. (b) Platform B.

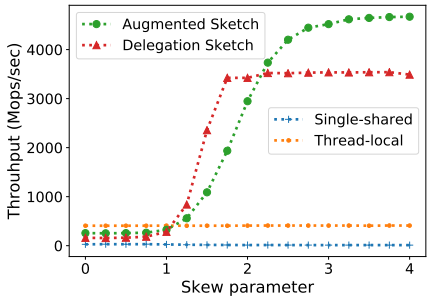
Figure 3.7: The effect of queries on the performance of the different parallelization designs, across two platforms. Both sets of experiments use data from the Zipf distribution with a skew parameter of 1.5.

36 threads).

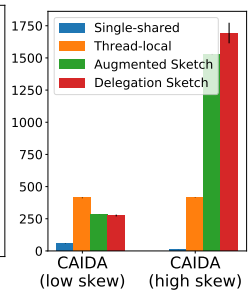
The same performance trend continues to hold on Platform B (Figure 3.6). The raw throughput achieved by each version is different, since this architecture has different characteristics (e.g. lower clock speed), but *Delegation Sketch* continues to outperform the baselines in all cases, especially with workloads that involve queries. While the performance of *Delegation Sketch* stops increasing when adding more than 150 threads in the 0.3% query workload, it is still more than 2 times faster compared to the baselines.

(B) Evaluating the effects of query rates

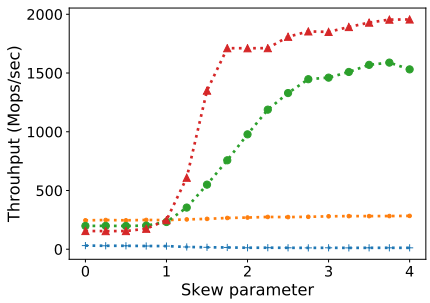
We now turn our attention to query rates and evaluate how they affect performance. In this experiment, we use all the available parallelism on each platform and plot the achieved throughput in Figure 3.7. For both platforms, increasing the rate of queries in the workload has no effect on the relatively low throughput of the single-shared design. Contrary, all other parallelization designs suffer a performance hit, even at a low query rate (0.1%). In the case of *Delegation Sketch*, this is because increasing the number of threads increases the number of filters that must be searched during a query. However, *Delegation Sketch* sustains an overall higher throughput than the baselines, because it avoids searching multiple sketches.



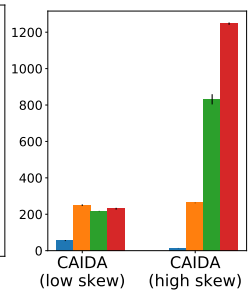
(a) 0% queries.



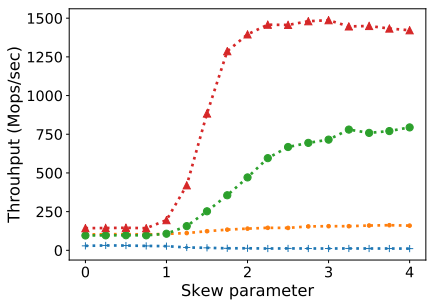
(b) 0.0% queries, real world data.



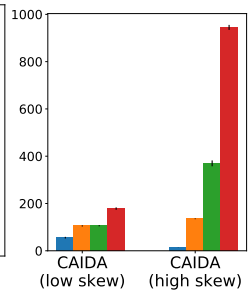
(c) 0.1% queries.



(d) 0.1% queries, real world data.



(e) 0.3% queries.



(f) 0.3% queries, real world data.

Figure 3.8: Platform A: Throughput comparison for different input skew and real data, using all the available threads (72). Note the different y-axis scale.

(C) Evaluating the effects of input skew

We now evaluate the effects of input skew on the performance of *Delegation Sketch*. In Figure 3.8, we present the throughput of all parallelization designs as we gradually increase the skew parameter of the distribution that generates the input keys, using three different query workloads. In the same figure, we also include the throughput achieved when using the two real world data sets we introduce in Section 3.7.1. We show the results of the execution in platform A and omit the results from platform B because they are equivalent.

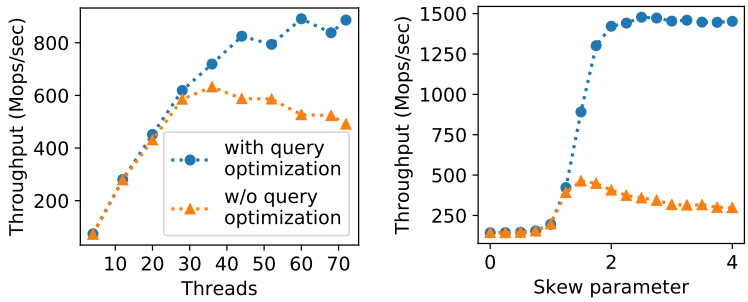
In general, Augmented Sketch and *Delegation Sketch* gain a dramatic increase in throughput when the skew parameter is more than 1.0. This is because both versions rely heavily on filters, that accelerate the processing of keys that are frequently found in the input. This result is in accordance with the experimental evaluation of [32]. At low skew (parameter values 0-1) the thread-local design that does not use filters outperforms all others, since in this case the filters only add overhead. For medium skew (parameter values 1-2), *Delegation Sketch* outperforms Augmented Sketch even if there are no queries. This is due to: (i) the use of more filters (T delegation Filters per sketch) and (ii) the domain splitting technique that reduces the range of keys that end up at each filter, making the input on that filter appear more skewed. At higher skew levels, most of the input stream is dominated by a few frequent elements. At this point, throughput stops increasing and Augmented Sketch outperforms *Delegation Sketch*. This is because, under such a high skewness, the per-key processing is so small that even the added overhead of computing $Owner(K)$ for *Delegation Sketch* becomes relatively significant. As expected, when we introduce queries in the workload, *Delegation Sketch* quickly outperforms the Augmented Sketch, even under high input skew (Figures 3.8c and 3.8e).

The same relative trends also hold with real-world data sets (Figures 3.8b, 3.8d and 3.8f). With the IP data set that exhibits low skew, the thread-local design outperforms the filter based ones in most cases, but *Delegation Sketch* performs better when using real-world data with high skew, especially at 0.3% query rates where it is more than 2 times faster than Augmented sketch and roughly 9 times faster than thread-local.

(D) Evaluating the effects of query squashing

We now evaluate the effect of Query Squashing separately. We compare the performance of *Delegation Sketch* to a modified version that does not include the optimization.

Figure 3.9a shows the scalability of both versions, in the same setting as the one used for Figure 3.5c. We see that, without the optimization, throughput



(a) Scalability.

(b) Effect of input skew.

Figure 3.9: The effect of Query Squashing, compared to a modified version that does not include it. Left: scaling with the number of threads for fixed input skew. Right: the effects of input skew, using 72 threads. In both cases the workload contains 0.3% queries.

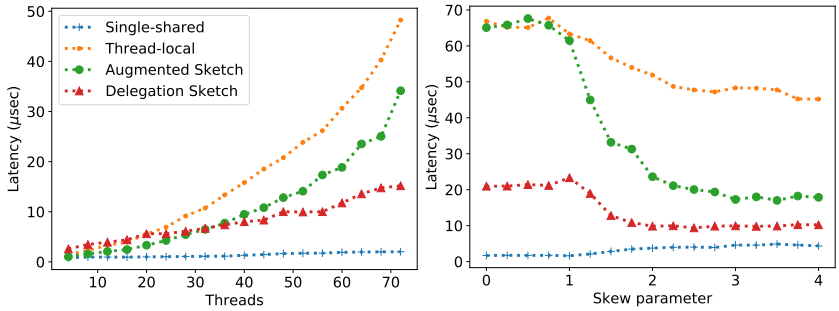
starts to drop after 20 threads and cannot scale to more than 36 threads. This is because, after that point, a large number of threads attempt to perform queries and as a result the query operation becomes a bottleneck, especially on the thread that is responsible for the most frequent key in the stream. At 72 threads, our optimization brings roughly 1.8X speedup in throughput.

The same effect holds when we increase the input skew of the stream. Since the keys we perform queries on come from the same distribution as the keys we insert (see Section 3.7.1), when skew is high, most threads try to query the same key K and have to wait for the thread $i = Owner(K)$ to handle them. Our optimization manages to overcome that bottleneck: by "squashing" all those queries into one operation, thread i is able to handle them all without repeatedly searching the filters and the sketch for the same key. At high skew (parameter value of 3.0), Query Squashing brings up to 4.5 times speedup in throughput, without introducing any overhead when the skew is low.

3.7.4 Query latency

So far, we evaluated performance based on the throughput of operations. We now take a closer look at the latency of queries across different versions.

In Figure 3.10a, we present the average latency of query operations depending on the number of threads, using data taken from the Zipf distribution with skew parameter 1.2. Overall, the single-shared sketch has extremely low query latency, less than 2.5 μsec, which only rises slightly at more than 36 threads. As expected, queries in the single-shared approach are very efficient since they only need to search for the key in a single sketch, albeit at the cost of low insertion



(a) Latency of queries as the number of threads increases.

(b) Latency of queries for different input skew.

Figure 3.10: Platform A: Latency of query operations as the number of threads and the input skew increase.

rate, as shown in Figure 3.5a. The latency of the thread-local design increases quickly with the number of threads, since the number of sketches that need to be searched increases. Augmented sketch has lower latency compared to the thread-local design, since some of the keys will be found in filters instead of sketches, but the overall latency remains high. *Delegation Sketch* manages to retain a lower latency than thread-local and Augmented Sketch under high parallelism (up to 2.25X and 3.18X times lower than the Augmented Sketch and thread-local respectively), since we search for a key in multiple filters but in at most one sketch.

Next, we fix the number of threads to 72 and vary the input skew. Again, the query latency of the single-shared approach is very low, but increases slightly under high skew, since the parts of the sketch where the common keys hash into become contention points. When the skew is low, the query latency of *Delegation Sketch* is up to 3X lower than that of Augmented Sketch and thread-local, since we avoid searching in multiple sketches. As the skew increases, the latency of approaches that use filters (Augmented Sketch and *Delegation Sketch*) is reduced, but *Delegation Sketch* manages to outperform Augmented Sketch, through the use of our Query Squashing optimization.

In the above comparisons, also note the following: a) Queries in *Delegation Sketch* occasionally perform extra work before they are completed e.g. helping insertions by flushing filters into sketches. That extra work is still taken into account in the query latency. b) As already mentioned in Section 3.7.1, the queries of Augmented Sketch are treated favorably since we do not introduce any synchronization.

3.7.5 Summary of the evaluation

In summary, we study how *Delegation Sketch* fares with respect to the following dimensions: accuracy, scalability, support for concurrent queries and skewed input. We showed that *Delegation Sketch* is as accurate (and often better) than the most accurate baseline with the same amount of memory, due to the Domain Splitting technique. At the same time, *Delegation Sketch* is highly scalable on both a NUMA multi-core and a massively parallel architecture.

The benefits of *Delegation Sketch* are more pronounced in the presence of concurrent queries, where *Delegation Sketch* achieves a higher relative speedup than the best performing baseline (up to 4X) and continue to hold the presence of highly skewed input distributions.

3.8 Related work

In this section, we present related work on sketches, the parallelization of sketches, and on systems that use them.

Sketches: The literature contains numerous sketch designs. Cormode [4, 6] present an overview of synopsis and sketch base approaches. We summarize the most relevant here. Apart from the Count-Min Sketch (summarized in Section 3.2), there are more sketches that follow similar techniques. The Count-Sketch [3] uses the median (rather than the minimum) of the hashed counters to provide an unbiased approximation (rather than an over-approximation). FCM [39] dynamically adjusts the number of counters that are incremented to reduce the approximation error on low frequency items. HeavyGuardian [42] separates the counter for high frequency items, while instead ColdFilter [48] uses a filter to count the frequencies of low-frequency items. The aforementioned sketches provide ingenious designs to optimize the space-accuracy-performance trade-off. *Delegation Sketch* is a generic parallelization design that is orthogonal to the underlying sketch, so it can be used to improve the parallel performance of the above mentioned sketches.

Parallelization: As already mentioned, most work in sketches focuses on the single thread scenario. The few papers that discuss parallelization of sketches suggest either a single-shared sketch or a thread-local approach. ElasticSketch [43] proposes a new sketch that is parallelized for multi-cores using a thread-local design. Alipourfard et al. [1] evaluate different measurement algorithms and show that the thread-local design has lower latency compared to the single-shared one. In contrast to our work, both of these do not consider continuous concurrent queries. Mandal et al. [27] use multiple copies that can be periodically merged into a single sketch, which improves the accuracy but requires a lot of commu-

nication between threads. FCM [39] uses multiple copies and splits the domain of keys to buckets of threads, similar to our *Delegation Sketch*, but their design is specific to the Cell processor that has a single centralized unit that distributes the stream to several co-processors.

Tangwongsan et al. [37] follow a single-shared sketch design and include algorithms for many query types, however their work lacks an experimental evaluation and comparison of their results. Das et al. [8] also keep a single shared summary and combine multiple operations, similar to our design. However, their approach is specific to the Space-Saving summary. Taşyaran et al. [38] use a single sketch but parallelize the hash computations among threads, at the cost of requiring frequent synchronization points. Roy et al. [32] propose a form of pipeline parallelism between two threads, where one thread is responsible for the filter while another handles the underlying sketch. Rinberg et al. [30,31] is, to our knowledge, the only other work that considers concurrent queries and insertions. They reduce the need for synchronization by introducing more relaxed semantics for sketches and rigorously prove correctness and linearizability with respect to those semantics. As they also explain, this relaxation comes at the cost of accuracy. Observing that our evaluation here shows that *Delegation Sketch* is as accurate as the best performing alternative, while maintaining scalability, it is worthwhile to note the interest in cross-studying further concurrency-aware consistency formulations.

Applications and Systems: Sketches have found uses in many applications, especially in traffic monitoring. These have often lead to the creation of monitoring systems that use sketch designs tailored for such applications, e.g., in software or hardware switches. UniMov [26] and HashPipe [33] are implemented for smart network cards and heavily rely on sketches. NitroSketch [25] builds on top of a fast packet I/O framework in the user-plane (DPDK) and supports high insertion speeds that reach line rate on 40Gbps links. Apart from traffic monitoring, Garofalakis et al. [13] use sketches to reduce state transfer in distributed monitoring system. However, none of those those systems discuss the trade-offs related to concurrent queries when parallelizing sketches.

3.9 Conclusions and future work

In this paper, we introduce *Delegation Sketch*, a generic parallelization design for sketches achieving high processing rates and accuracy in the presence of concurrent queries. We analyze existing parallelization designs and find they cannot support both insertions and queries efficiently and have to prioritize one or the other. Contrary to this, *Delegation Sketch* reduces the cost of queries

while at the same time maintains a high insertion rate through: (i) maintaining multiple parallel sketches, (ii) splitting the domain of keys to different sketches and ensuring that all occurrences of the same key end-up in the same sketch, thus making queries faster and (iii) using multiple filters that aggregate occurrences of the same key locally. By combining multiple concurrent queries of the same key, *Delegation Sketch* significantly reduces the cost of queries under high input skew.

We thoroughly evaluate *Delegation Sketch* across multiple dimensions, namely: accuracy, scalability, query rate and input skew. We use two massively parallel platforms and experiment with both real and synthetic data. Our evaluation shows that *Delegation Sketch* is able to scale up to more than 72 threads, especially in the presence of concurrent queries, achieving from 2.5X to 4X higher throughput than the best performing baseline, while being as accurate as the most accurate baseline. These results contribute to efficient and accurate stream processing, especially for applications that require frequent, responsive queries to dynamically changing data streams, e.g., software switches that schedule traffic dynamically or spatio-temporal monitoring applications and more [11, 14, 35, 46]. Extending the design with support for more types of queries, for even higher impact on such applications is an interesting future direction. It is also interesting to consider a co-design of *Delegation Sketch* together with efficient concurrent implementations of the underlying sketches (e.g. using concurrent counters [18, 40, 41]), as well as a distributed deployment across multiple nodes. The source code of *Delegation Sketch* is available online: `https://github.com/mpastyl/DelegationSketch`.

Acknowledgements

We would like to thank the anonymous EuroSys reviewers and our shepherd Aleksandar Prokopec for their valuable feedback that helped improve the paper. The research leading to these results has been partially supported by the Swedish Civil Contingencies Agency (MSB) through the projects RICS and RIOT, by the Swedish Foundation for Strategic Research (SSF) through the framework project FiC, by the Swedish Research Council (VR) through the project ChaosNet and the project AgreeOnIT, the Vinnova-funded project "KIDSAM", and from the European Community's Horizon 2020 Framework Programme under grant agreement 773717.

Bibliography

[1] Omid Alipourfard, Masoud Moshref, Yang Zhou, Tong Yang, and Minlan Yu. A comparison of performance and accuracy of measurement algorithms in software. In *Proceedings of the Symposium on SDN Research*, SOSR '18, pages 18:1–18:14, New York, NY, USA, 2018. ACM.

[2] Arm. ODROID-XU3. `https://developer.arm.com/graphics/development-platforms/odroid-xu3`. Accessed: 2019-11-04.

[3] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, ICALP '02, pages 693–703, Berlin, Heidelberg, 2002. Springer-Verlag.

[4] Graham Cormode. Sketch techniques for approximate query processing. *Foundations and Trends in Databases. NOW publishers*, 2011.

[5] Graham Cormode. *Count-Min Sketch*, pages 464–468. Springer New York, New York, NY, 2016.

[6] Graham Cormode, Minos Garofalakis, Peter J Haas, Chris Jermaine, et al. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends® in Databases*, 4(1–3):1–294, 2011.

[7] Graham Cormode, Theodore Johnson, Flip Korn, S. Muthukrishnan, Oliver Spatscheck, and Divesh Srivastava. Holistic UDAFs at streaming speeds. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 35–46, New York, NY, USA, 2004. ACM.

[8] Sudipto Das, Shyam Antony, Divyakant Agrawal, and Amr El Abbadi. Thread cooperation in multicore architectures for frequency counting over multiple data streams. *Proc. VLDB Endow.*, 2(1):217–228, August 2009.

[9] Alin Dobra, Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Processing complex aggregate queries over data streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 61–72, New York, NY, USA, 2002. ACM.

[10] Romaric Duvignau, Marina Papatriantafilou, Konstantinos Peratinos, Eric Nordström, and Patrik Nyman. Continuous distributed monitoring in the evolved packet core. In *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*, pages 187–192, 2019.

[11] Zhang Fu, Magnus Almgren, Olaf Landsiedel, and Marina Papatriantafilou. Online temporal-spatial analysis for detection of critical events in cyber-physical systems. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 129–134. IEEE, 2014.

[12] Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. *Data Stream Management: A Brave New World*, pages 1–9. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[13] Minos Garofalakis, Daniel Keren, and Vasilis Samoladas. Sketch-based geometric monitoring of distributed stream queries. *Proc. VLDB Endow.*, 6(10):937–948, August 2013.

[14] Vincenzo Gulisano, Yiannis Nikolakopoulos, Ivan Walulya, Marina Papatriantafilou, and Philippas Tsigas. Deterministic real-time analytics of geospatial data streams through scalegate objects. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pages 316–317, 2015.

[15] Red Hat. Receive Packet Steering (RPS). `https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/network-rps`, 2019. Accessed: 2019-10-22.

[16] Red Hat. Receive-Side Scaling (RSS). `https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/network-rss`, 2019. Accessed: 2019-10-22.

[17] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, page 355–364, New York, NY, USA, 2010. Association for Computing Machinery.

[18] Maurice Herlihy, Beng-Hong Lim, and Nir Shavit. Scalable concurrent counting. *ACM Transactions on Computer Systems (TOCS)*, 13(4):343–364, 1995.

[19] Qun Huang, Xin Jin, Patrick P. C. Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. Sketchvisor: Robust network measurement for software packet processing. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 113–126, New York, NY, USA, 2017. ACM.

[20] Intel. Knights Landing. `https://ark.intel.com/content/www/us/en/ark/products/codename/48999/knights-landing.html`. Accessed: 2019-10-30.

[21] Jiawei Jiang, Fangcheng Fu, Tong Yang, and Bin Cui. Sketchml: Accelerating distributed machine learning with data sketches. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1269–1284, New York, NY, USA, 2018. ACM.

[22] Leslie Lamport. On interprocess communication. part i: Basic formalism. *Distributed Computing*, 1(2):77, 1985.

[23] Leslie Lamport. On interprocess communication, part ii: Algorithms. *Distributed Computing*, 1:86–101, 1986.

[24] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 311–324, Santa Clara, CA, March 2016. USENIX Association.

[25] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pages 334–350, New York, NY, USA, 2019. ACM.

[26] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 101–114, New York, NY, USA, 2016. ACM.

[27] Ankush Mandal, He Jiang, Anshumali Shrivastava, and Vivek Sarkar. Topkapi: Parallel and fast sketches for finding top-k frequent elements. In *Proceedings of the 32Nd International Conference on Neural Information Processing Systems*, NIPS'18, pages 10921–10931, USA, 2018. Curran Associates Inc.

[28] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. AGILE: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proceedings of the 10th International Conference on Autonomic Computing ({ICAC} 13)*, pages 69–82, 2013.

[29] Yiannis Nikolakopoulos, Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. A consistency framework for iteration operations in concurrent data structures. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 239–248. IEEE, 2015.

[30] Arik Rinberg, Alexander Spiegelman, Edward Bortnikov, Eshcar Hillel, Idit Keidar, Lee Rhodes, and Hadar Serviansky. Fast concurrent data sketches. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, page 117–129, New York, NY, USA, 2020. Association for Computing Machinery.

[31] Arik Rinberg, Alexander Spiegelman, Edward Bortnikov, Eshcar Hillel, Idit Keidar, and Hadar Serviansky. Fast concurrent data sketches. *CoRR*, abs/1902.10995, 2019.

[32] Pratanu Roy, Arijit Khan, and Gustavo Alonso. Augmented sketch: Faster and more accurate stream processing. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1449–1463, New York, NY, USA, 2016. ACM.

[33] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 164–176, New York, NY, USA, 2017. ACM.

[34] The CAIDA UCSD anonymized internet traces - 2018. `http://www.caida.org/data/passive/passive_dataset.xml`, 2019. Accessed: 2019-11-03.

[35] Charalampos Stylianopoulos, Magnus Almgren, Olaf Landsiedel, and Marina Papatriantafilou. Continuous monitoring meets synchronous transmissions and

in-network aggregation. In *2019 15th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 157–166. IEEE, 2019.

[36] Yongmin Tan, Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Chitra Venkatramani, and Deepak Rajan. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*, pages 285–294. IEEE, 2012.

[37] Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. Parallel streaming frequency-based aggregates. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 236–245, New York, NY, USA, 2014. ACM.

[38] Fatih Tasyaran, Kerem Yildirir, Kamer Kaya, and Mustafa Kemal Tas. One table to count them all: Parallel frequency estimation on single-board computers. *CoRR*, abs/1903.00729, 2019.

[39] Dina Thomas, Rajesh Bordawekar, Charu C. Aggarwal, and Philip S. Yu. On efficient query processing of stream counts on the Cell processor. In *2009 IEEE 25th International Conference on Data Engineering*, pages 748–759, March 2009.

[40] Junchang Wang, Tao Li, and Xiong Fu. Accurate counting algorithm for high-speed parallel applications. *Concurrency and Computation: Practice and Experience*, 31(13):e5090, 2019.

[41] Roger Wattenhofer and Peter Widmayer. The counting pyramid: an adaptive distributed counting scheme. *Journal of Parallel and Distributed Computing*, 64(4):449–460, 2004.

[42] Tong Yang, Junzhi Gong, Haowei Zhang, Lei Zou, Lei Shi, and Xiaoming Li. Heavyguardian: Separate and guard hot items in data streams. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, pages 2584–2593, New York, NY, USA, 2018. ACM.

[43] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 561–575, New York, NY, USA, 2018. ACM.

[44] Tong Yang, Lingtong Liu, Yibo Yan, Muhammad Shahzad, Yulong Shen, Xiaoming Li, Bin Cui, and Gaogang Xie. Sf-sketch: A fast, accurate, and memory efficient data structure to store frequencies of data items. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 103–106, April 2017.

[45] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 29–42, Lombard, IL, 2013. USENIX.

[46] Nikos Zacheilas, Vana Kalogeraki, Yiannis Nikolakopoulos, Vincenzo Gulisano, Marina Papatriantafilou, and Philippas Tsigas. Maximizing determinism in stream processing under latency constraints. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, pages 112–123, 2017.

[47] Yang Zhou, Peng Liu, Hao Jin, Tong Yang, Shoujiang Dang, and Xiaoming Li. One memory access sketch: A more accurate and faster sketch for per-flow measurement. In *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, pages 1–6, Dec 2017.

[48] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. Cold filter: A meta-framework for faster and more accurate stream processing. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 741–756, New York, NY, USA, 2018. ACM.

Part III

Fast and Energy-Efficient Processing on Embedded Accelerators

PAPER III

Charalampos Stylianopoulos, Linus Johansson,
Oskar Olsson, Magnus Almgren

CLort: High Throughput and Low Energy Network Intrusion Detection on IoT Devices with Embedded GPUs

Adapted version of the paper that appeared in *the Proceedings of the 23rd Nordic Conference on Secure IT Systems (NordSec)*, Secure IT Systems, pp. 87–202, LNCS vol. 11252, Springer 2018.

4

CLort: High Throughput and Low Energy Network Intrusion Detection on IoT Devices with Embedded GPUs

Abstract

While IoT is becoming widespread, cyber security of its devices is still a limiting factor where recent attacks (e.g., the Mirai bot-net) underline the need for countermeasures. One commonly-used security mechanism is a Network Intrusion Detection System (NIDS), but the processing need of NIDS has been a significant bottleneck for large dedicated machines, and a show-stopper for resource-constrained IoT devices. However, the topologies of IoT are evolving, adding intermediate nodes between the weak devices on the edges and the powerful cloud in the center. Also, the hardware of the devices is maturing, with new CPU instruction sets, caches as well as co-processors. As an example, modern single board computers, such as the Odroid XU4, come with integrated Graphics Processing Units (GPUs) that support general purpose computing. Even though using all available hardware efficiently is still an open issue, it has the promise to run NIDS more efficiently.

In this work we introduce *CLort*, an extension to the well-known NIDS Snort that i) is designed for IoT devices ii) alleviates the burden of pattern matching for intrusion detection by offloading it to the GPU. We thoroughly explain how our design is used on top of Snort and suggest various optimizations to enable processing on the GPU. We evaluate *CLort* in regards to throughput, packet drops in Snort, and power consumption using publicly available traffic traces. *CLort* achieves up to 52% faster processing throughput than its CPU counterpart. *CLort* can also analyze up to 12% more packets than its CPU counterpart when sniffing a network. Finally, the experimental evaluation shows that *CLort* consumes up to 32% less energy than the CPU counterpart, an important consideration for IoT devices.

4.1 Introduction

Even though Internet of Things (IoT) technologies have become widespread and mature, cyber security is still a problem. Several attacks, across very different environments, demonstrate in painstaking detail that the community needs to build security mechanisms suitable for IoT, or else deployment may slow down. A recent example is the series of attacks against the electricity network in both the distribution and transmission grid in Ukraine by controlling the devices found in substations.

Challenges to improve security in IoT stem from different factors. For a long time, an IoT system was designed with very limited edge devices that

communicated with a powerful cloud. Even though the cloud could handle many security mechanisms, the attacks happen at the edges of the network, targeting devices that need to be cheap, conserve power and are too limited to run their own security mechanisms. Fortunately, modern IoT systems have become more heterogeneous with different types of devices. The previously limited edge is becoming slightly more powerful with new processors and architectures, and the powerful cloud has been complemented by a range of devices, the so-called fog in-between the edge and the cloud, with devices that offer more computational power and, for some applications, a much faster response rate than sending the data to the cloud. These intermediate IoT devices promise to also improve the security of the system as a whole.

In this paper, we take advantage of the recent maturity of IoT devices and investigate how a network intrusion detection system, one of the cornerstones of regular IT security, can run efficiently in the IoT. More specifically, as recently released devices come with integrated co-processors or graphics processing units, we investigate how to use the full hardware of a dedicated "security node" to improve the speed (throughput) of the analysis, while using less energy to do so. Moreover, as one challenge of IoT is the distributed nature of the system, it may not be possible to define a single choke point for network analysis. As we demonstrate that our solution processes packets faster, it may be possible to run the intrusion detection system on existing nodes in the network while still leaving enough CPU cycles for the nodes' primary function.

The outline of the paper is the following. In Section 4.2, we outline background concepts related to this work, namely Snort, the Aho-Corasick algorithm and a high-level description of general purpose computing on GPUs. In Section 4.3, we explain the design of our system followed by the evaluation in Section 4.4. Section 4.5 describes related work and we conclude the paper in Section 4.6.

4.2 Background

Given the prominence of Snort as a network intrusion detection system, we start with an introduction to such systems in general and Snort in particular. We then describe the pattern matching algorithm in Snort (Aho-Corasick). Finally, we give a brief background on general purpose computing on GPU devices.

4.2.1 Network Intrusion Detection Systems and Snort

The purpose of a Network Intrusion Detection System (NIDS) is to inspect all incoming and outgoing network traffic and alert for any malicious behaviour. Many NIDS are *signature-based*, meaning that they rely on a set of patterns that are part of known attacks or vulnerabilities. One of the benefits of NIDS, over for example a firewall, is that they inspect not only the packet headers but also the packet payload (a.k.a. *deep packet inspection*) in order to detect a wide range of malicious attacks.

Nowadays, Snort is one of the most commonly deployed signature-based NIDS. Originally developed in the late 90s, Snort has been in active development ever since and has become the de facto NIDS. Its most recent version (Snort 3, in alpha version when this paper is written), offers many new features, such as a modular architecture, cross-platform support and multi-threaded processing of traffic from different interfaces.

Snort relies on *rules* that determine what kind of malicious behaviour it should look for in a packet. Rules usually contain a fixed string pattern, as well as other options that need to be true to flag a packet as malicious (e.g., traffic towards specific ports). A very brief outline of Snort's processing pipeline is the following: (i) Snort *captures* packets from a network interface or a capture file, (ii) a *decode* module creates common metadata for this packet, such as source and destination ports and encapsulated protocols, (iii) packets that belong to a TCP stream are reassembled, (iv) a *search engine* performs pattern matching on the packets, where the payload data are compared against the malicious patterns, and (v) if a match is found, a *validation* step is invoked to ensure that the rest of the rule options are also true for the packet containing the match. Finally, (vi) Snort outputs a verdict for the packet (whether or not it is malicious).

The pattern matching in step (iv) is an expensive bottleneck and therefore the focus of this paper. Snort uses the Aho-Corasick pattern matching algorithm, as described below.

4.2.2 The Aho-Corasick patten matching algorithm

The Aho-Corasick algorithm [1][1] is a popular, *state machine* based algorithm that allows Snort to match the payload against multiple patterns at the same time. The first step of Aho-Corasick is to build a state machine out of all the patterns, where the individual characters in the patterns become the transitions

[1] For short it will be simply called Aho-Corasick in the following sections.

to new states. The state machine is usually implemented as a two-dimensional *state transition array*, with a row for each state and a column for every possible transition from that state to the next one. An extra bit in the array is reserved for final states, i.e. states that indicate that a full pattern has been matched.

After building the state machine at setup time, performing pattern matching on the packet payload is relatively straightforward: starting from the initial state, the algorithm examines one character and uses it to determine the next state. The algorithm keeps jumping from state to state, based on the information found in the state transition array. If the execution reaches one of the final states, a pattern has been found in the payload and Snort will then check other parameters of the full rule before sending out an alert.

We have chosen to use Aho-Corasick as a cornerstone for the work in this paper because: (i) it is what Snort actually uses and (ii) it can be parallelized, making it a good match for the GPU.

4.2.3 General Purpose GPU Computing

Originally designed for graphics processing tasks, in the last decade GPUs have been proven increasingly successful for offloading computation from the CPU [5]. Hence, General Purpose Computing on the GPU (GPGPU computing) is a term used for the use of GPUs to perform tasks that would be usually performed on the CPU.

The internal architecture of GPUs involves thousands of threads (orders of magnitudes more than on a standard CPU) that have a very simple pipeline and generally operate on a lower frequency. As such, the GPU is an appealing platform for computing tasks that benefit from a high degree of parallelization.

There are two main frameworks that make general purpose computing possible on GPUs: CUDA [10], developed by NVIDIA and OpenCL [8], an open-source library developed by the Khronos Group. Although high-end desktop GPUs have been extensively used for various projects using these two frameworks, embedded GPUs, such as the one we use in this project, have only recently gained support for GPGPU computing. The platform used in this work offers OpenCL 1.2 support, so we use this framework in this paper.

4.3 Design of *CLort*

As one of the most expensive operations of the NIDS for the CPU is the pattern matching engine, we describe the design of *CLort* and the way it extends Snort by offloading the pattern matching to the GPU. We start with the general, high level

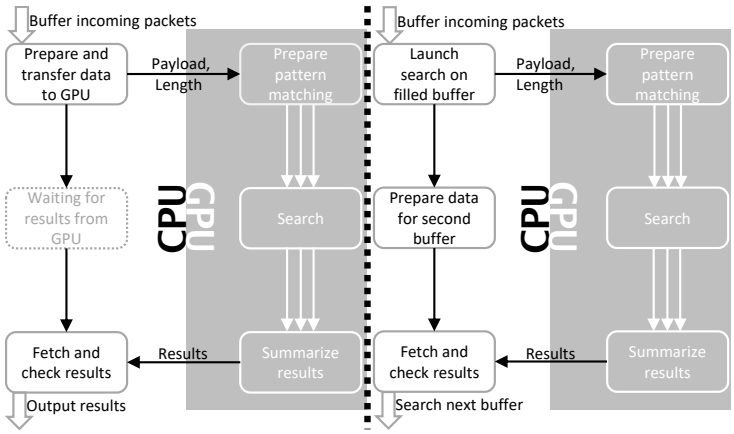


Figure 4.1: The left part shows the high level design of *CLort*, with the different steps involved in offloading the pattern matching of Snort to the GPU. The right part depicts an optimization with double buffering to increase the utilization of the CPU.

design of *CLort*. Then, we discuss issues related to several steps of this design, namely the transferring of data to and from the GPU and the parallelization of pattern matching on the GPU. Finally, we show how optimizations, such as the double buffering technique, are incorporated into our design to get the most speedup.

4.3.1 *CLort*'s general design

The general design of *CLort* is described in the left part of Figure 4.1 (where the right part is described later in Section 4.3.4). Incoming packets enter *CLort*'s pipeline after being processed by the first, pre-processing stages of Snort (see Section 4.2.1). The payload of each packet is sent to the GPU, to be checked against the state machine created by the patterns that are relevant to that packet. After that, the GPU executes the kernel that implements the Aho-Corasick pattern-matching algorithm. The CPU waits until the execution of the GPU is finished and the results are available. After that, execution continues with the rest of Snort's pipeline that includes validating the matches and logging the verdict for the packet (i.e. logging whether it is malicious).

4.3.2 Data transfers between the CPU and the GPU

Performing the pattern matching on the GPU requires that relevant data is transferred to the memory of the GPU, and then that the result is transferred back to the CPU. In general, data transfers to and from the GPU's device memory can be a significant bottleneck. However, for our hardware (further described in Section 4.4.1), the particular characteristics of the GPU offer an interesting way to alleviate that bottleneck. The Mali GPU of the Odroid XU4 does not have a separate device memory but shares the physical memory with the CPU. Thus, we can avoid unnecessary data transfers by mapping the memory region (using OpenCL's interface) of the data that we should send. The memory region is then directly accessible to the GPU. To allow the CPU to read the results, we map the region back to the CPU address space.

Related to data transfers, it is worth mentioning some details on the data structures that are transferred (or, in our case, mapped) to the GPU, specifically the state machine of Aho-Corasick (described in Section 4.2.1). Originally, the state machine is a two-dimensional array, with a row for each state and a column for each possible transition from that state to a next one. Here we note that: (i) in order to be mapped to the GPU, the state machines need to be serialized as a one-dimensional array (a simple transformation). The serialization and the corresponding mapping of the memory only happen once per state machine during setup, as the state machines are read-only data structures known at the start of Snort. (ii) Snort creates multiple state machines based on traffic characteristics (protocols, ports, etc.) and packets are matched against a state machine that is relevant to their traffic which also our implementation respects: when a packet is mapped to the GPU for processing, the correct state machine is used as an argument to the kernel that will process that packet.

4.3.3 Search on the GPU: Parallel Aho-Corasick

When state machines and the packet payloads are available to the GPU, pattern matching is performed using the Aho-Corasick algorithm (Section 4.2.2).

We parallelize Aho-Corasick in the following way: we split the payload data into a number of chunks, equal to the number of available GPU threads. Each thread is able to process its own chunk, in parallel, without the need for inter-thread communication. The input is divided evenly, so that every thread has equal amount of work to do, compared to the other threads. This avoids the problem of some threads terminating early and stalling, which exists in other parallelization methods for Aho-Corasick [9].

However, splitting the payload into chunks might result in a malicious pattern

being split across more than one chunk, with no single thread being able to detect the full pattern in "their" part. In order to detect such patterns, we let each thread process a fixed number of characters also from the chunk of the next thread (equal to the length of the longest pattern). This way, every malicious pattern will be detected by at least one thread. The disadvantage, however, is that short patterns that exist at the beginning of the chunks will be reported by two threads. We compensate by keeping an auxiliary data structure that holds the length of every pattern that is associated with a final state (a state indicating that a full pattern has been found). When we have a match in a thread, we use this data structure to determine the starting position of the match. If the start is within the chunk of the thread that found the match, it will be reported otherwise it will be ignored (as the next thread "owning" that chunk will find the same pattern and report it).

4.3.4 Packet Buffering: the double-buffering technique

As mentioned in other work [7, 18], launching a kernel for every single packet is not efficient for two main reasons. Firstly, there is significant overhead associated with launching a GPU kernel and it is good to amortize this cost over several packets. Secondly, with a single packet, especially if the packet is small, there might not be enough parallelism to fully exploit the GPU. There will not be enough data to distribute to all available GPU threads or each thread will only process a very small amount of data before exiting. For that reason, we buffer packets on the CPU to submit in batches to the GPU. When a new packet arrives in the Snort pipeline, it will be copied into a buffer. The processing of that packet is postponed at this point and Snort can continue acquiring new packets. When the buffer is full, we launch the GPU kernel to process all packets at once. Having more data to process allows us to make the most of the parallelism the GPU has to offer. Even though we introduce a small amount of latency before a packet is being processed, it is not a problem on regular networks as the buffer is significantly smaller than the traffic received during a short period of time. However, as we describe later in Section 4.4, our current implementation that uses buffers cannot make use of the final parts of Snort's pipeline (validation and verdict).

We have investigated two different designs in our work (Figure 4.1). In the basic design (to the left in the figure), when a kernel is being executed on the GPU, the CPU waits until the end of the execution to get the results. While this is a straightforward design, it does not optimize throughput for a node dedicated for monitoring the network but may work well if there are other tasks needing cycles on the CPU.

In the *double buffering* design (shown to the right), both the CPU and the GPU perform work in parallel and, as will be shown in our evaluations, this increases the utilization of the CPU. In short, in the double buffering technique, as proposed by [19], two buffers are used to store packets on the CPU. When the first buffer is full and the GPU starts processing packets, the CPU can keep buffering packets in the second buffer. When the second buffer is also full, the CPU will first collect the results from the GPU execution, before launching another kernel to process data in the second buffer. Thus, the double buffering technique helps hide the latency of executing the kernel by pipelining that execution with the buffer generation and result collection.

In Section 4.4.2 we measure the effect of the double buffering technique and show that it successfully reduces the overall processing time.

4.4 Evaluation

We implemented *CLort* using the OpenCL framework. This section presents the results from the experimental evaluation of *CLort*, using a wide range of experiments to measure and evaluate the benefits that *CLort* brings in intrusion detection for IoT. The experiments are performed on four versions of Snort: Snort original, Snort modified (CPU), *CLort* single buffer (GPU), and *CLort* double buffer (GPU). The *Snort modified (CPU)* is included to make the comparisons as fair as possible. This version of Snort behaves just like *CLort* (buffers packets and does not perform the validation and verdict steps from Section 4.2.1), but runs the search on the CPU. All comparisons and relative speedups reported use Snort modified (CPU) as a baseline.

4.4.1 Experimental methodology

Hardware: We use the Odroid XU4 platform [12], a single board computer with a big.LITTLE architecture (ARM Cortex-A15 and ARM Cortex-A7). The reason for choosing this hardware platform is that it supports an integrated GPU (ARM Mali-T628, 6 shader cores) that is compatible with OpenCL 1.2, allowing us to perform General Purpose Computing on its GPU. The GPU offers many interesting differences compared to standard high-end GPUs, such as individual program counters for each thread, the lack of local memory, as well as a shared device memory between the GPU and CPU (2GB). The device also supports a high speed Ethernet port, making it a good candidate for a high speed NIDS. For a subset of the experiments (c.f. Section 4.4.4) an almost identical platform is used (Odroid XU3), that, contrary to the XU4, is equipped with energy sensors

Name	Details
SmallFlows	Appneta sample, 9.4 MB data, 1209 flows over a 5 minute duration.
BigFlows	Appneta sample, 368 MB data, 40686 flows over a 5 minute duration.
ISCX12 131	The first 1 million packets from ISCX2012 on 13 of June, 634 MB of data from a data set that includes activity from network infiltration.
ISCX12 121	The first 1.5 million packets from ISCX2012 on 12 of June, 1.01 GB of data from a data set without malicious activity.
ISCX12 12 Full	The entire file from ISCX2012 on 12 of June, 4.22 GB of data from a data set without malicious activity.

Table 4.1: The data sets used throughout the evaluation section.

but with a slower network card.

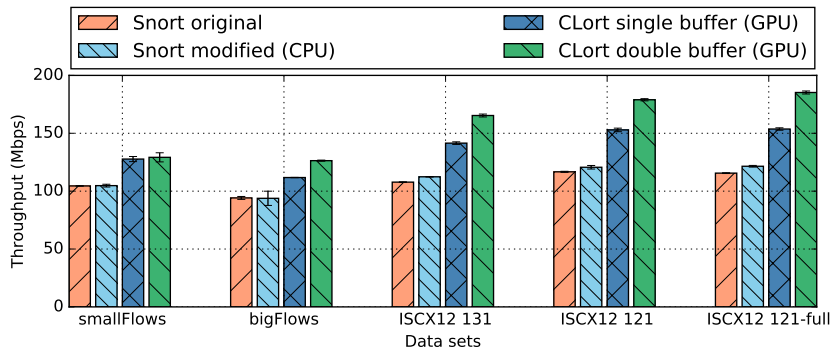
The Odroid would most likely be counted as quite powerful for consumer IoT in the home, but its cost could be motivated for professional settings for industrial IoT, especially if the node can run several functions for the network. Moreover, accounting for the recent trends of development of the hardware (i.e. Raspberry Pie 3), it is likely that these devices will also be common in the consumer space.

Realistic Traffic Traces: We use publicly available data sets that capture a realistic behaviour of network traffic for the experiments in this paper. Five different capture files are used, as shown in Table 4.1. The first two traffic traces (hereby named *SmallFlows* and *BigFlows*) come from Appneta [2], the current developers of Tcpreplay. SmallFlows is a synthetic capture representing a combination of different applications and BigFlows is a capture of real traffic from a busy private network.

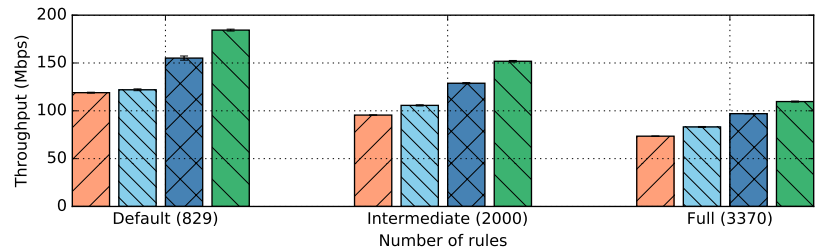
The other capture files come from ISCXIDS2012 [15, 16]. These data sets are specifically designed to simulate real traffic in order to test and evaluate IDSs. These capture files are larger, ranging from just a few up to several gigabytes. As all capture files are publicly available, they form a repeatable baseline.

Rule sets: Unless otherwise stated, we use the 829 rules (each rule containing at least one pattern) that are enabled by default in Snort's community distribution. In Section 4.4.2, we experiment with bigger sets of rules.

Metrics: First, we measure the *throughput*: how much traffic is processed per unit of time (Section 4.4.2). We then measure the *percentage of received packets that are analyzed* by the NIDS (either Snort or *CLort*), when capturing live traffic from the network interface (Section 4.4.3). We also measure the



(a) The overall throughput of *CLort*, across different data sets.



(b) The overall throughput of *CLort*, for different numbers of rules

Figure 4.2: Throughput evaluation of *CLort* across different (a) data sets and (b) number of rules.

power consumption (important consideration for IoT devices): what is the power consumption of different hardware components when processing incoming traffic (Section 4.4.4).

4.4.2 Evaluating throughput

The first set of experiments focus on *throughput*, by varying the traffic to be analyzed as well as the number of rules in Snort.

(A) Overall throughput:

Figure 4.2a presents the processing throughput across different data sets, where we measure the complete execution of Snort (Section 4.2.1) by reading the

pcap files from disk. In these experiments, we use the default number of rules (829 rules). The experiments were repeated 5 times and we report the average and the standard deviation of the measured throughput across all 5 runs.

First, both *CLort* versions that use the GPU consistently outperform the CPU versions across all data sets in our experiments, suggesting that the GPU is capable of accelerating the task of pattern matching. We achieve up to 52% higher throughput compared to the CPU (modified) version of Snort, which is significant, considering that: (i) we only offload pattern matching (step iv) from Section 4.2.1, while the other steps of Snort's processing (steps i-iii) are still part of the measured time and (ii) we achieve it using resources (the embedded GPU) that are already available on the platform.

Second, in almost all cases, the double buffering technique provides a performance boost (up to 20%) compared to the single buffer approach. This means that the double buffer optimization successfully overlaps the CPU and GPU execution, keeping both processing units busy with useful work.

(B) Varying the number of rules:

By changing the number of rules, we can determine how it affects the Snort runtime performance for scenarios with more rules than the default community rule set (baseline, 829 rules). We enable all available rules that contain fixed string patterns (3370 rules) and also create an intermediate set with 2000 randomly chosen rules.

We run the experiments with several pcap files from Table 4.1, but only include the ISCX12 121 data set as the results were similar across all runs. In Figure 4.2b, both *CLort* versions that utilize the GPU continue to outperform the CPU versions of Snort. Increasing the number rules reduces the raw throughput of all versions as expected since the state machines grow larger and there is extra processing work for the rest of Snort's pipeline. In the case of the full rule set, we see that the relative speedup achieved by *CLort* is smaller. This is because many of the extra rules introduce processing that is not related to the search engine that we parallelize (e.g. many of the rules involve regular expression matching).

4.4.3 Sniffing the network

The experiments in Section 4.4.2 show that *CLort* has a higher processing throughput when reading packets from a capture file. In this section, we test the performance of *CLort* in a setting much closer to the way a NIDS is deployed in practice by capturing traffic directly from the network.

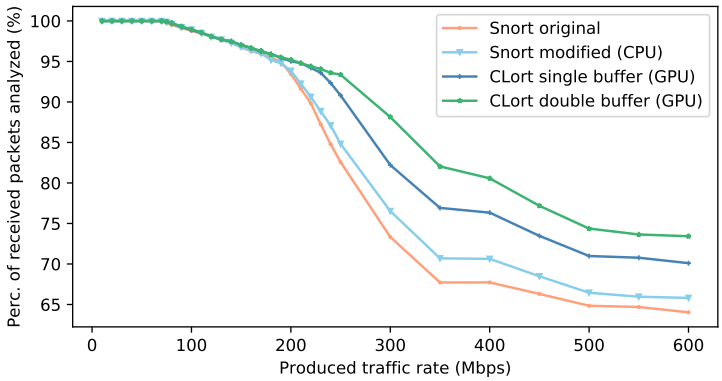


Figure 4.3: Percentage of the received packets that *CLort* managed to analyze, as we increase the rate at which we replay traffic.

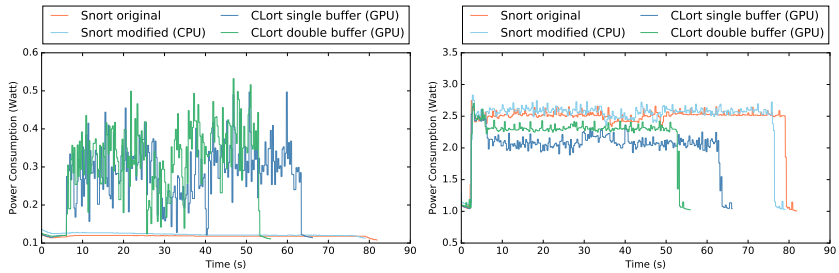
The experimental setup is the following. We connect the Odroid XU4, running *CLort*, to the span port of a switch (HP V1910-24G). As such, it sees all traffic on the network segment handled by the switch. We then use a laptop (MacBook Pro '14) to replay the pcap files from the ISCX12 131 data set using *tcpreplay* at different speeds. Also, versions of Snort and *CLort* use the default set of 829 rules. The network segment also contains a *dhcp* server, so there is spurious minimal traffic in addition to the traffic being replayed by the laptop.

There are several potential bottlenecks in the system: the hardware replaying the pcap file, the switch handling the span port, the network card of the Odroid in promiscuous mode, the kernel processing before handing the packets to the NIDS, and finally the NIDS's pipeline. To exclude problems beyond our improvements of Snort, we measure the ratio between the packets that are received by the NIDS and the ones that the NIDS successfully analyzes.

Figure 4.3 shows the percentage of the received packets that *CLort* and Snort manage to analyze at various traffic rates. After approximately 70Mbps, all versions start dropping packets. However, both versions of *CLort* are able to process a larger portion of the received packets, up to 12% more than the modified CPU version of Snort. These results show that the throughput gained from using the GPU translates to *CLort* being able to handle more packets than its CPU counterpart.

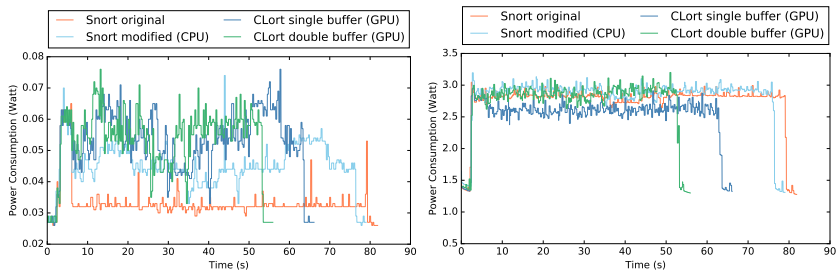
4.4.4 Evaluating energy consumption

The final part of the evaluation studies the energy consumption. The ODROID-XU4 is unfortunately not equipped with power measuring sensors. For this



(a) Power consumption on GPU

(b) Power consumption on CPU(a15)



(c) Power consumption on RAM

(d) Total power consumption

Figure 4.4: Power consumption measurements of the CPUs, GPU and RAM.

reason, we use an older version (ODROID-XU3 [11]) for the energy consumption experiments. The ODROID-XU3 is equipped with the same processor setup as well as the *same GPU and CPU* as the ODROID-XU4. The only significant difference (for the power consumption tests) between these two hardware systems is that the network card is slower for the XU3 (100 Mbps instead of 1Gbps), but the RAM speed and the memory bandwidth is faster. The RAM speed of the ODROID-XU3 is 933Mhz and the memory bandwidth is 14.9GB/s, whereas the RAM speed of the ODROID-XU4 is 750Mhz and the memory bandwidth is 12GB/s.

We measured the power consumption of the following three components: CPU (A15), GPU and RAM memory with a sample rate of 100 samples/second running the ISC12 121 data set using the default number of rules. Figure 4.4 summarizes the results, with one graph each for the CPU, GPU, RAM, along with the total power consumption. Note that each sub-figure uses its own scale on the y-axis.

As expected, looking at Figure 4.4a (the power consumption of the GPU in isolation), we can see that only the GPU versions consume any power on

Version	Average Power (W)	Total Energy Consumed (Joule)
CLort GPU (double)	2.87	**145.7**
CLort GPU (single)	**2.63**	159.4
Snort CPU (modified)	2.91	215.6
Snort CPU (original)	2.83	217.5

Table 4.2: Average power draw and total energy consumed for each version.

the GPU, while the CPU versions consume little to no power on the GPU. The double buffer version of *CLort* consumes slightly more power than the single buffer, but for a shorter period of time.

The power consumption of the CPU (A15) in Figure 4.4b shows that the CPU versions are almost equal in their execution time and they consume the most power. The GPU versions utilize the CPU less; since the pattern matching has been offloaded to the GPU, it leads to lower power consumption on the CPU. The single buffer version consumes the least CPU power on average between the different versions (close to 2 Watt) but runs longer than the double buffer version.

Figure 4.4c shows the power consumed by the memory, where the range on the y-axis is very small compared to the other components. In general, the memory is responsible for only a small part of the power draw in all versions, never more than 0.08 W. Notice that the original version of Snort consumes the least amount of power on average. This is because all other versions include extra memory operations to read and write packet data to the buffers.

Figure 4.4d shows the total, aggregated power consumption from the different components. Overall, the CPU versions of Snort and the double buffer version of *CLort* have almost the same average power draw, though the double buffer version has a much shorter execution time. The single buffer GPU version consumes the least amount of power (2.63 Watt on average).

Table 4.2 summarizes the average power consumption, along with the total energy consumed during the execution time of each version. The single buffer version of *CLort* consumes 9.8% less power on average than the CPU version making it a better fit for scenarios where the power envelope is limited. On the contrary, the double buffer version of *CLort* consumes less energy in total (32.4% less than the CPU), since it is able to process traffic faster. This, and in conjunction with the results from Section 4.4.3 makes it an appealing alternative for scenarios where the traffic load is high and the total consumed energy must be minimized.

4.5 Related work

Below we discuss related work, divided into two lines of work: NIDS on high-end systems with GPUs and then NIDS on devices typical of IoT.

4.5.1 NIDS on GPUs

Over the years, significant efforts have focused on accelerating the functions of a NIDS using high-end, desktop GPUs. The seminal work by Jacob et al. [6] was the first to offload the pattern matching on the GPU. Due to the lack of general-purpose GPU programming APIs at the time, they used graphics libraries (OpenGL). Their prototype, PixelSnort, achieved at best a 40% increase in performance when the CPU was under high load, but with no noticeable performance gain under normal load. Moreover, their pattern matching algorithm is based on the Boyer-Moore algorithm [3], which evaluates each pattern individually, making it hard to scale for a large number of patterns.

More recent work takes advantage of the ease of programming and performance offered by general purpose APIs such as OpenCL and CUDA. Vasiliadis et al. [18] use CUDA and implement the Aho-Corasick algorithm to offload pattern matching and Xie et al. [20] use OpenCL to implement a modified version of Aho-Corasick (PFAC [9]). Apart from differences with our design, both of these works target high-end GPUs, while we focus on resource-constrained, embedded GPUs that share resources with the CPU (memory).

Another, interesting line of work focuses on how to make efficient use of all the computing devices in the system and orchestrate the processing between the CPU and the GPU. Vasiliadis et al. [19] present Midea, a system based on Snort that makes use of highly parallel CPUs, multiple GPU devices and networks cards. They also describe different optimization techniques to alleviate bottlenecks, due to data transfers and synchronization. Jamshed et al. [7] present Kargus, a similar, highly parallel system based on their own, custom IDS. Recently, Papadogiannaki et al. [13] presented a scheduler that dynamically distributes the packet processing workload across a system with heterogeneous hardware resources (including both discrete and integrated GPUs). Finally, Go et al. [4] also show that integrated GPUs are a cost-effective alternative for packet processing. All the above-mentioned work achieve very high processing throughput using high-end CPUs and GPUs and target large-scale networks or even backbone traffic. Contrary, we focus on resource-constrained devices that better fit the area of IoT networks.

4.5.2 NIDS on IoT related devices

Security for IoT and resource-constrained devices is an active research topic. A project that examines the feasibility of using Snort for resource-constrained devices, similar to the spirit of this work, is RPiDS by Sforzin et al. [14]. In this work, a Raspberry Pi 2 running Snort to function as a portable IDS was thoroughly tested to evaluate the capacity of modern single-board-computers. The measurements showed that the Raspberry Pi could run Snort without ever filling its entire memory capacity. These results strengthen the argument that single-board-computers are a reasonable choice for security in future IoT networks, especially since it is expected that hardware improves with time. However, when the authors experimented with live traffic they reported that there are packet losses, even at low rates, which we also confirm in our experiments (Section 4.4.3). This raises interesting questions on the bottlenecks involved in the system that cause these losses. In this work, we take one step further and show how more hardware feature available at these devices (e.g. the GPU) can be used to improve the feasibility of a NIDS on resource-constrained devices and reduce the above-mentioned packet losses.

Moving to even more low-end devices and cyber-physical systems, a large body of work focuses on custom IDS that are tailored to the functionality of such devices. One such example is Tabrizi et al. [17] that present a software tool, which produces a customized IDS based on the memory capacity of the targeted device. Given the user-defined security coverage functions, the security properties of the system and memory requirements, the tool can produce an IDS customized to operate on the specified system. The authors were able to produce an IDS, tailored for an electrical smart meter, that operated on 4MB of memory. However, different from this work, they propose an anomaly-based IDS and their main focus is on minimizing memory consumption for low-end devices.

4.6 Conclusions

In this paper, we consider the security of the Internet-of-Things and address the processing challenges that are part of Network Intrusion Detection Systems. Specifically, we propose *CLort*, a system based on the latest release of Snort (version 3.0) that is designed to tackle the processing needs of NIDS for high-end IoT devices by offloading pattern matching to a GPU. We describe the system design and the effects of various optimizations, such as a double-buffering technique.

We thoroughly evaluate the performance of *CLort* under realistic traffic and show that by using the GPU: (i) *CLort* achieves up to 52% faster processing

throughput than Snort (ii) is able to process up to 12% more packets from the network interface under high load and, (iii) achieves the above while consuming 32% less energy than its CPU counterpart.

The work in this paper suggests a hardware-aware design that uses the GPU capabilities offered by modern, high-end IoT devices is an appealing alternative that strengthen security by alleviating the processing bottlenecks of security countermeasures, such as network intrusion detection. The source code of *CLort* is available at `https://github.com/Arklights/Master`

Acknowledgements

The research leading to these results has been partially supported by the Swedish Civil Contingencies Agency (MSB) through the project "RICS" and by the European Community Horizon 2020 Framework Programme through the UNITED-GRID project under grant agreement 773717. We also thank Simon Kindström for his help with the energy measurements.

Bibliography

[1] Alfred V. Aho and Margaret J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM*, 18(6):333–340, June 1975.

[2] Appneta. Sample captures. `http://tcpreplay.appneta.com/wiki/captures.html/` [Accessed: 2018-09-18].

[3] Robert S. Boyer and J. Strother Moore. A Fast String Searching Algorithm. *Commun. ACM*, 20(10):762–772, October 1977.

[4] Younghwan Go, Muhammad Asim Jamshed, YoungGyoun Moon, Changho Hwang, and KyoungSoo Park. Apunet: Revitalizing GPU as packet processing accelerator. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 83–96, Boston, MA, 2017. USENIX Association.

[5] GPGPU. General-Purpose Computation on Graphics Hardware. `http://gpgpu.org` [Accessed: 2018-07-19].

[6] Nigel Jacob and Carla Brodley. Offloading IDS Computation to the GPU. In *22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 371–380, Dec 2006.

[7] Muhammad Asim Jamshed, Jihyung Lee, Sangwoo Moon, Insu Yun, Deokjin Kim, Sungryoul Lee, Yung Yi, and KyoungSoo Park. Kargus: A Highly-scalable Software-based Intrusion Detection System. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 317–328, New York, NY, USA, 2012. ACM.

[8] Khronos group. OpenCL Overview. `https://www.khronos.org/opencl/` [Accessed: 2018-07-19].

[9] Cheng-Hung Lin, Chen-Hsiung Liu, Lung-Sheng Chien, and Shih-Chieh Chang. Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs. *IEEE Transactions on Computers*, 62(10):1906–1916, Oct 2013.

[10] NVIDIA. About CUDA. `https://developer.nvidia.com/about-cuda` [Accessed: 2018-07-19].

[11] ODROID-XU3. ODROID-XU3. `http://www.hardkernel.com/main/products/prdt_info.php?g_code=g140448267127` [Accessed: 2018-06-08].

[12] ODROID-XU4. ODROID-XU4 User Manual. `https://magazine.odroid.com/wp-content/uploads/odroid-xu4-user-manual.pdf` [Accessed: 2018-03-28].

[13] Eva Papadogiannaki, Lazaros Koromilas, Giorgos Vasiliadis, and Sotiris Ioannidis. Efficient software packet processing on heterogeneous and asymmetric hardware architectures. *IEEE/ACM Transactions on Networking*, 25(3):1593–1606, June 2017.

[14] Alessandro Sforzin, Félix Gómez Mármol, Mauro Conti, and Jens-Matthias Bohli. RPiDS: Raspberry Pi IDS - A Fruitful Intrusion Detection System for IoT. In *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld), Toulouse, France, July 18-21, 2016*, pages 440–448, 2016.

[15] Ali Shiravi, Hadi Shiravi, Mahbod Tavallaee, and Ali A. Ghorbani. Intrusion detection evaluation dataset (ISCXIDS2012). `http://www.unb.ca/cic/datasets/ids.html` [Accessed: 2018-05-08].

[16] Ali Shiravi, Hadi Shiravi, Mahbod Tavallaee, and Ali A. Ghorbani. Toward developing a systematic approach to generate benchmark datasets for intrusion detection. *Computers & Security*, 31(3):pp. 357–374, 2012.

[17] Farid Molazem Tabrizi and Karthik Pattabiraman. Flexible Intrusion Detection Systems for Memory-Constrained Embedded Systems. In *2015 11th European Dependable Computing Conference (EDCC)*, pages 1–12. IEEE, Sept 2015.

[18] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *Recent Advances in Intrusion Detection: 11th International Symposium, RAID 2008, Cambridge, MA, USA, September 15-17, 2008. Proceedings*, pages 116–134, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[19] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. Midea: A multi-parallel intrusion detection architecture. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, New York, NY, USA, 2011. ACM.

[20] Hongying Xie, Yangxia Xiang, and Caisen Chen. Parallel Design and Performance Optimization based on OpenCL Snort. In *Proceedings of the 2017 2nd Joint International Information Technology, Mechanical and Electronic Engineering Conference, JIMEC*, 2017.

PAPER IV

Charalampos Stylianopoulos, Simon Kindström, Magnus Almgren,
Olaf Landsiedel, Marina Papatriantafilou

Co-Evaluation of Pattern Matching Algorithms on IoT Devices with Embedded GPUs

Adapted version of the paper that appeared in *the Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*, pp. 17–27, ACM 2019.

5

Co-Evaluation of Pattern Matching Algorithms on IoT Devices with Embedded GPUs

Abstract

Pattern matching is an important building block for many security applications, including Network Intrusion Detection Systems (NIDS). As NIDS grow in functionality and complexity, the time overhead and energy consumption of pattern matching become a significant consideration that limits the deployability of such systems, especially on resource-constrained devices. On the other hand, the emergence of new computing platforms, such as embedded devices with integrated, general-purpose Graphics Processing Units (GPUs), brings new, interesting challenges and opportunities for algorithm design in this setting: how to make use of new architectural features and how to evaluate their effect on algorithm performance. Up to now, work that focuses on pattern matching for such platforms has been limited to specific algorithms in isolation.

In this work, we present a systematic and comprehensive benchmark that allows us to co-evaluate both existing and new pattern matching algorithms on heterogeneous devices equipped with embedded GPUs, suitable for medium- to high-level IoT deployments. We evaluate the algorithms on such a heterogeneous device, in close connection with the architectural features of the platform and provide insights on how these features affect the algorithms' behavior. We find that, in our target embedded platform, GPU-based pattern matching algorithms have competitive performance compared to the CPU and consume half as much energy as the CPU-based variants. Based on these insights, we also propose *HYBRID*, a new pattern matching approach that efficiently combines techniques from existing approaches and outperforms them by 1.4x, across a range of realistic and synthetic data sets. Our benchmark details the effect of various optimizations, thus providing a path forward to make existing security mechanisms such as NIDS deployable on IoT devices.

5.1 Introduction

With the widespread adoption of Internet of Things (IoT) technologies, an increasing number of devices are equipped with the ability to communicate and connect to the Internet. While promising increased efficiency and flexibility, connected devices are vulnerable, as shown by recent attacks that specifically targeted IoT devices such as connected cameras and thermostats [31, 42]. Protecting such devices with well-established security mechanisms such as network intrusion detection systems (NIDS) is necessary. Yet, such mechanisms are hard to deploy on these devices, because their core function depends on pattern matching, a bottleneck that needs significant resources (more than 70% of the

running time of the system may be spent on pattern matching [2]).

In the context of NIDS, pattern matching algorithms scan the packet payload (Deep Packet Inspection) and detect any occurrence of malicious string signatures (known in advance) in the stream of packets. Pattern matching algorithms are often studied in close relation with the hardware platforms, because the characteristics of the target platform play an important role on performance. This is evident by the number of algorithms in the literature that target specific platforms or build on optimizations that utilize specific characteristics, e.g., CPU caches [10], vector instructions [37], FPGAs [36] or hardware accelerators [21].

In IoT deployments, one can find significant hardware diversity from the edge to the core of the network. For example, the introduction of new computing approaches with new hardware diversity in the fog increases the design space for pattern matching algorithms and offers new capabilities for improvements of performance. This is leveraged in recent research [32] with algorithms tailored to medium-ranged embedded platforms, such as Raspberry Pi or Odroid [6]. The latter platform offers an interesting combination of an IoT board equipped with an embedded, programmable Graphics Processor Unit (GPU). Such medium/high range IoT devices can take the role of NIDS boxes, protecting a network of resource-constrained devices closer to the edge of that network.

However, challenges remain and the feasibility of such platforms for NIDS is not established yet, especially with respect to performance, since their hardware characteristics are different from the well-studied high-end platforms. First, significant effort is required to take an algorithm for a particular hardware and change it to run well on another type of hardware. Given the development effort, it would be favorable to better understand what algorithms to port and how. Secondly, it is not clear how different optimizations translate across hardware. For example, the design of many pattern matching algorithms (e.g. [10, 37, 41]) is driven by specific features of the target architecture, such as cache sizes and vector execution units, so it is unclear how they perform on a different system. Thirdly, most work is documented in isolation, with specific data, and within a specific framework.

As such, a common methodology for benchmarking algorithms tailored for fog-layer devices is needed, to understand possibilities and limitations of the hardware itself, as well as the effects of algorithm engineering and, most importantly, the interplay of the algorithm and the hardware features.

Contributions: In this work we present a co-evaluation of various pattern matching algorithms on a heterogeneous computing platform. We target a medium range embedded device (an Odroid XU3) that is equipped with a programmable GPU, i.e., a GPU that can support general-purpose computing.

- We design a benchmark that facilitates a systematic comparison between

different pattern matching algorithms, using realistic data sets that capture real NIDS workloads.

- We co-evaluate well-known CPU and GPU based algorithms, including our own GPU adaptation of a state of the art CPU based algorithm (DFC).

- We evaluate the effect of different platform-specific optimizations and parameters in the performance of the algorithms, both in terms of execution time as well as energy consumption, to guide the community in future research.

- Based on our methodology, we were also able to create a new algorithm, *HYBRID*, that effectively combines the benefits of existing approaches and achieves up to 1.4x speedup in pattern matching compared to the best GPU baseline.

The remainder of the paper is organized as follows: in Section 5.2 we discuss the general aim and design considerations of the benchmark. Section 5.3 summarizes the algorithms included in our benchmark, both existing as well as new ones. Section 5.4 provides details on the target hardware and discusses relevant optimizations. In Section 5.5 we present and discuss the benchmark results. We present related work in Section 5.6 and conclude in Section 5.7.

5.2 Benchmarking aim and considerations

This section discusses the high-level design considerations for our benchmark and serves as a guideline for the general methodology followed in our work. We motivate the aim of the benchmark, the choice of algorithms and the steps we consider towards a fair and useful comparison between them.

Utilization of the target platform. The aim of this benchmark is to analyze the performance of pattern matching algorithms on a specific set of newly introduced hardware platforms: embedded devices with integrated GPUs that support General Purpose GPU computing (GPGPU). Originally designed for processing graphics, in the last decade, GPUs have been proven particularly successful in accelerating general-purpose workloads as well, mostly due to their highly parallel architecture. Their popularity increased further with the introduction of libraries that simplify the writing of GPU programs, namely CUDA [28] and OpenCL [16]. We focus on algorithms that are written or can be ported to OpenCL, since it is the library that the hardware supports (c.f. Section 5.4).

Choice of algorithms. Each pattern matching algorithm follows a different approach, but most of them fall into two main categories: *state machine* based approaches and *filtering* based ones. The first category involves variants of the Aho-Corasick algorithm (c.f. next section), where a state machine is created out of the patterns and traversed based on the input. On the contrary, filtering based approaches try to quickly isolate parts of the input that do not contain any matches and spend more resources on the parts that might potentially match with one of the patterns.

In the benefit of covering a wide spectrum of approaches and gaining insights from how different algorithms perform on the device we target in this work, we pick representative algorithms from both families. The description of the chosen algorithms follows in the next section.

General and platform-specific parameters and characteristics. The design and performance of the algorithms mentioned above are greatly affected by the system parameters and the characteristics of the target architecture. Each family of algorithms is affected by those characteristics to a different extent. Thus, it is important to examine the effect of a variety of parameters on algorithms from different families, especially when targeting architectures with unique characteristics such as the one we use in our work. In Section 5.4 we discuss the architecture characteristics and the parameters that are relevant to examine.

Use of realistic data sets and patterns. The performance of pattern matching algorithms for NIDS is often highly data-dependent and fluctuates based on i) the number and size of patterns used, and ii) the type of traffic that is monitored, e.g., randomly generated data versus captured traffic from actual deployments. Knowing the type of traffic and how the performance of each algorithm will be affected beforehand is hard. However, it is important to experiment with sets of traffic and patterns that are as close to the ones found in real life as possible. We use publicly available data sets that simulate traffic from real deployments, as well as patterns taken directly from the official pattern distributors for Snort [35], the de-facto NIDS. We refer to Section 5.5 for more information on the choice of data sets and patterns.

Identical functionality. Putting different algorithms together under the same fair and meaningful framework is not trivial. Often, algorithms are designed with different requirements in mind, e.g., reporting all the matches and their positions in the input, versus just reporting how many matching patterns the input contains. It is hard to judge which functionality to implement, since different applications of those pattern matching algorithms might have different requirements. However, it is important to ensure that we keep the same functionality for the algorithms we compare in this benchmark. For this reason, we have

manually inspected the compared versions and rewritten parts of them to ensure that the different implementations have identical functionality.

In later sections we will return to the considerations mentioned above and discuss some of their implementation details.

5.3 Considered algorithms & novel designs

Since this benchmark focuses on NIDS applications of pattern matching, we consider algorithms that have been designed or used in NIDS workloads. As previously mentioned, we cover algorithms from the two main pattern matching algorithm families: *state machine* based and *filter* based.

The algorithms chosen from the literature are representative of methods that can benefit from different architectural features of the computing platform, such as the high parallelism/latency hiding of the GPU as well as cache memories. Based on insights from the existing approaches and their properties in heterogeneous computing platforms, we also introduce a new method, which we call *HYBRID*, that aims to combine the benefits of the two. This algorithm will also be part of our benchmark. The algorithms are summarized in Table 5.1, where we also specify the code repository (if applicable) as well as the effort to adapt the code to the evaluation.

5.3.1 State machine based algorithms: Aho-Corasick and Parallel Failure-less Aho-Corasick

One of the most well-known algorithms for pattern matching is Aho-Corasick (AC) [1]. Aho-Corasick has a preprocessing stage where it builds a *Finite State Automaton (FSA)*, in other words a state machine, with all patterns. However the FSA differs from a normal FSA as *failure transitions* are also added. These failure transitions occur when there is a mismatch between the input and the state machine, and point to the state sharing the longest common prefix with the current state. During processing, one character at a time is read from the input and used to determine the next state, using a state transition table that represents the state machine. An example of Aho-Corasick's state machine is shown in Figure 5.1. The arrows between each branch indicate failure transitions.

Aho-Corasick is a simple way of performing multiple pattern matching that requires a small number of operations per byte. However, storing all the states and their transitions requires significant memory [26, 27]. Because of the large memory requirements, many cache misses occur during state transitions [26]. Nonetheless, AC is often used in practice and a variant of AC is used in NIDS

Acronym	Algorithm	Family	Code	Programming Effort	Comment
AC (CPU)	Aho-Corasick [1]	state machine	Snort repo [34]	low	CPU baseline, used in Snort
DFC (CPU)	Direct Filter Classification [10]	filter	[10,37]	low	CPU baseline (filter-based)
PFAC (GPU)	Parallel Failureless AC [21]	state machine	[4]	medium	code required some work to adapt to benchmark
DFC (GPU)	Direct Filter Classification [37]	filter	[37]	high	our own algorithm implementation (the first implementation of DFC for the GPU) based on [37]
HYBRID (GPU)	Mix of DFC and PFAC	mixed	this paper	high	our own design: a hybrid combining DFC and PFAC, targeting the GPU

Table 5.1: A summary of the evaluated algorithms and their acronyms.

such as Snort [35]. We include Aho-Corasick as a CPU-based baseline in our benchmark, due to its widespread use.

Parallel Failure-less AC (PFAC) is a parallel implementation of Aho-Corasick with a focus on GPUs [21]. In PFAC, every GPU thread starts from a single character and follows the state transitions until a match is detected or the state machine has returned to the original state, indicating there was not a match starting from that character. The state machine in PFAC is simplified compared to that of Aho-Corasick in that the failure transitions are removed and each pattern is an individual branch in the state machine. PFAC spawns many threads (up to one thread per input character) but most of them will quickly detect no matches and exit. Algorithm 5.1 shows a highly simplified pseudo-code version of PFAC.

One reason that makes PFAC an interesting algorithm to include in this benchmark is the fact that it has been evaluated on resource-constrained em-

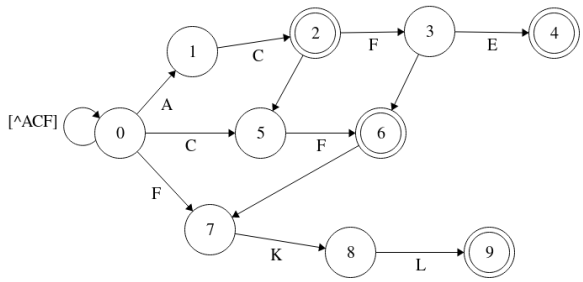


Figure 5.1: Aho-Corasick state machine for the patterns AC, ACFE, CF and FKL.

bedded GPUs, similar to the ones we target in this paper. Aragon et al. [4] implement PFAC in OpenCL and evaluate it on an ARM Mali GPU, considering various optimizations. In this work, we build upon this evaluation and extend it with more algorithms and insights.

5.3.2 Filter based algorithms: DFC and V-Patch

The motivation behind filter based algorithms is to create small filters that can quickly determine and discard parts of the input that do not contain any matches, in a quick and cache-efficient way, contrary to the cache-inefficient data structures of state machine based algorithms.

Direct Filter Classification (DFC) is a state of the art, memory and cache efficient pattern matching algorithm implemented on CPUs, using Direct Filters (DFs) [10]. A Direct Filter is a bitmap that summarizes some consecutive bytes from the pattern, and is small enough to be cache resident. A 2 byte DF will use 2 consecutive bytes from a pattern, e.g., the first or last two bytes, to index a bit in the bitmap.

DFC performs matching in multiple phases: *filtering* and *verification*. In the filtering phase, a window of two bytes is slid over the input, summarized and matched to the filter. If the window matches, additional DFs requiring more bytes

Algorithm 5.1. High level pseudo-code of PFAC.

```
1 for each character C in input stream do
2 |  find the first state based on character C
3 |  while no patterns found AND state not reset to 0 do
4 |  |  read next character
5 |  |  traverse the state machine
6 |  end
7 end
```

Algorithm 5.2. High level pseudo-code of DFC.

```
1 for each character C in input stream do
2 |  feed C (and neighbouring character) through a series of filters
3 |  if there is a hit in the filters then
4 |  |  do verification
5 |  end
6 end
```

for indexing may be used to check that it really is a match. The verification phase performs exact matching using a compact hash table with efficient indexing. Unlike other filter based algorithms (e.g. FFBF [26]), DFC works with patterns of any length and avoids expensive hash computations for indexing the filters. Algorithm 5.2 shows a highly simplified pseudo-code version of DFC.

Since its inception, DFC has been the basis for further improvements on its filter design and its ability to utilize features of the architecture, such as vectorization. Stylianopoulos et al. [37] re-design the filter architecture of DFC (S-Patch) to better fit realistic traffic scenarios and allow for a vectorizable design (V-Patch), which leads to an increased throughput of up to 3.6x compared to the original DFC algorithm.

In this work, we implemented our own GPU version of DFC. We based our filter design on the CPU filtering design of S-Patch [37]. Such a GPU implementation is not straight-forward and involves re-factoring the CPU version and orchestrating the communication between the CPU and the GPU. We briefly mention here that the data structures that hold the series of hash tables used in the verification version of DFC need to be serialized so that they can be transferred to the GPU. Moreover, the results of the pattern matching kernel on the GPU are stored in a buffer that indicates whether or not there was a match, for each character in the input. Those results are later transferred to the CPU and processed there.

Algorithm 5.3. High level pseudo-code of *HYBRID*.

```
1  for each character C in input stream do
2  |  feed C (and neighbouring character) through a series of filters
3  |  if there is a hit in the filters then
4  |  |  find the first state based on character C
5  |  |  while no patterns found AND state not reset to 0 do
6  |  |  |  read next character
7  |  |  |  traverse the state machine
8  |  |  end
9  |  end
10 end
```

We include in our experiments both the CPU version of DFC, as a baseline, as well as our own GPU algorithm implementation. Upon these implementations, we evaluate and discuss the effect of different optimizations and algorithm engineering methods.

5.3.3 A hybrid approach

In addition to algorithms from the two different families described above, in this work we also introduce a new approach, *HYBRID*, that borrows the benefits of both families.

The new, albeit simple idea behind this approach is to filter the input using one filter, similar to the ones used in DFC and then perform PFAC-based pattern matching only on the parts of the input that cause a hit in the filters. The motivation behind this scheme is that it combines: (i) the good cache locality of filter based approaches, allowing us to quickly filter out most of the input (c.f. Section 5.5.5) and (ii) the ability to avoid the costly verification part of DFC by falling back to PFAC which uses only a minimum number of operations (jumps from one state to the other). The fact that, in PFAC, we can traverse the automaton starting from any individual character in the input (contrary to, e.g., Aho-Corasick), makes it possible to pipeline the execution of PFAC with that of DFC, by starting an automaton traversal only where there is a hit in the filter. Algorithm 5.3 shows a highly simplified pseudo-code version of *HYBRID*. The *HYBRID* algorithm is a result of the insights gained from the benchmark described in this paper.

5.4 Hardware-oriented algorithm optimizations

In this section, we provide details on the architecture of the target platform and discuss relevant algorithm engineering methods that make use of the architectural features.

5.4.1 Overview of the target platform

We use the ODROID-XU3 [6] to execute all tests. The XU3 uses the Exynos 5 Octa (5422) chip that has a quad-core ARM Cortex-A15 and quad-core ARM Cortex-A7, along with 2GB of RAM. The Exynos 5422 is used in one variant of the Samsung Galaxy S5, showing that the XU3 is a reasonable choice for a more powerful embedded device today. It also suggests that hardware and the pattern matching algorithms considered in this paper could also be used, e.g., in scanning malicious mobile applications. The most important reason as to why we use the XU3 is because it possesses a GPU that allows General-Purpose computing on Graphics Processing Units (GPGPU). Further reasons are that it has a high-speed Ethernet port, allowing for high-speed network sniffing. We chose the Odroid XU3 over the newer XU4 model (that has the same CPU and GPU) because the former has on-board energy sensors that allow us to easily measure the energy consumed.

The XU3's variant of the GPU is a Mali-T628 MP6 [5] that has 6 cores (much less than the typical high-end discrete GPUs). These shader cores may be programmed using OpenCL. The GPU does not have a separate device memory but share the same physical memory as the CPU [7]. Moreover, any shared or local memory on the GPU is actually mapped in the global memory instead. Each core has L1 and L2 memory caches to remedy the cost of always accessing the global memory. These caches have a 64-byte cache line. There are two 16KB L1 caches for each core, one used for generic memory accesses and one for texture memory. Another unique feature of the Mali GPU is that there is Single Instruction Multiple Data (SIMD) parallelism supported within each GPU thread. Finally, each GPU thread has a separate instruction counter, meaning that divergent execution is not a problem on the Mali-GPU [7]. Architectures with similar features (e.g. shared memory between the CPU and the GPU) are also available from NVIDIA [29] and we expect similar performance trends.

Based on the above the description, the Odroid XU3 platform is a relatively powerful single board computer that belongs to the medium/high range of IoT devices. Although the architecture and the capabilities of the device are far different from those of the typical end-point, resource-constrained sensors (that are usually powered by ARM Cortex-M or similar processors), medium/high

Configurations			
Acronym	Description	Effort	Comment
MAP	The use (or not) of memory mapping to avoid data transfers.	low	Done through the OpenCL API
THR	Thread granularity.	low	Done through the OpenCL API
MEM	Type of memory used to store the filters of DFC (GLOBAL/LOCAL/TEXTURE).	high	Using texture memory requires data structure rearrangement
VEC	The use (or not) of vectorization in the GPU kernel.	high	Vectorization requires code refactoring
WG	Work group size.	low	Done through the OpenCL API

Table 5.2: A list of optimizations and their acronyms used throughout the evaluation.

range devices have a central role in the IoT context, as they can play the role of cluster-heads or gateways to a network of less powerful end-nodes. Also, platforms like the Odroid XU3 fit in the fog computing [11] context, as an intermediate layer between the cloud and the edge network.

5.4.2 Relevant algorithm optimizations

In this section, we discuss optimizations that are relevant to explore and relate to (i) features of the architecture and (ii) possibilities for the evaluated algorithms to make good use of those features. A similar discussion on some of these parameters can also be found in the work by Aragon et al. [4]. We extend their discussion with more parameters to also consider the use of texture memory for storing the relevant data structures, as well as the use of vectorization, as it has shown promise for CPU-based algorithms [37]. In Section 5.5.2 we evaluate the effect of each of the optimizations described in detail in the following sections. They are also summarized in Table 5.2, with a comment on how they were realized and the corresponding effort.

Reducing memory transfers (MAP): Memory transfers between an OpenCL host (CPU) and OpenCL device (GPU) is often the bottleneck in GPGPU applications [13]. In most cases, memory transfers include a copy of data from the physical memory of the host to that of the GPU. The platform we target here offers an interesting way to alleviate this problem. The GPU shares the same physical address spaces as the CPU, making memory copies redundant. A naive use of the OpenCL API would still force the driver to perform memory copies

of the data to be transferred. Instead, it is possible to map a memory region (through the OpenCL API) to make it accessible from the GPU and then map it back to the CPU to read the results.

Increasing work per thread (THR): An issue with having each thread handle a single character, is that many threads are spawned. This is not a free operation, costing time and energy. By having each thread process multiple characters, fewer threads are needed.

Utilizing local memory (MEM LOCAL/GLOBAL): The GPU local memory is shared between each work-item in a workgroup and is often faster to access than the global memory. As described earlier, the target platform does not have dedicated local memory and references to local memory are served by the global memory instead. Nonetheless, we experiment with local memory to show how optimizations that would be beneficial on high-end GPUs can actually have a negative impact on embedded platforms like ours.

Utilizing texture memory (MEM TEXTURE): An additional optimization strategy used in this work is to utilize the texture memory, an on-chip memory designed to quickly serve addresses that have spatial locality. As there is a separate L1 cache for textures (16KB) in the XU3, one may increase the cache hits further by storing one or more DF as a texture. It is worth noting that storing the filters in texture memory results in the need for some additional registers and computations to retrieve the one bit of interest, potentially reducing the gain from better cache locality.

Vectorized execution (VEC): As mentioned earlier, each GPU thread can operate on multiple elements at a time in a SIMD fashion. In this work we implement and evaluate a vectorizable version of DFC, based on V-Patch [37], and *HYBRID*. In this version, the filtering is done on multiple (in this case eight) elements at the same time. Notice however that some parts of the filtering are still done in scalar code, due to the lack of special vector instructions such as the gather instruction [25], which is important for the implementation of V-Patch.

Altering OpenCL workgroup size (WG): Another variable is the size of the OpenCL workgroup [7]. Fewer workgroups should result in a lower overhead for maintaining them, but more results in better latency hiding. The size of a workgroup is usually a maximum of 256, as is the case on the ODROID-XU3.

5.5 Evaluation

In this section we co-evaluate the algorithms presented in Section 5.3 under the same benchmark. We start by describing the methodology followed throughout the experiments. Then we focus on DFC (which has not been previously studied

in a GPU context) and study the effects of different optimizations. We then report and discuss the overall performance comparison between the different versions, as well as how the performance changes across different data sets and number of patterns. Finally, we discuss some insights from the execution of *HYBRID*.

5.5.1 Evaluation methodology

In our study we use the Odroid XU3, presented in Section 5.4, to run all experiments. We test the algorithms using the three following publicly available data sets that represent realistic traffic traces:[1]

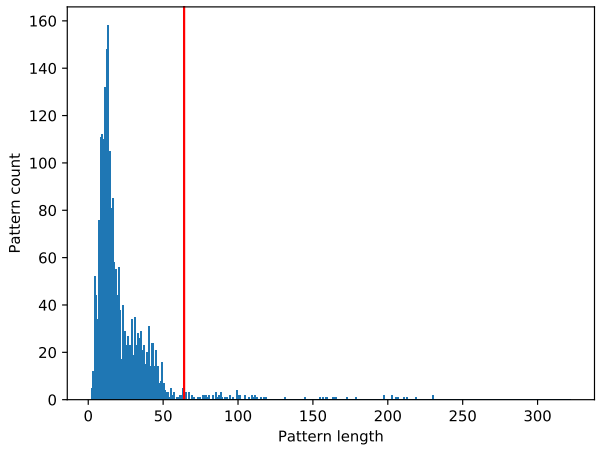
- 162MB of HTTP traffic from the DARPA 2000 data set [20] (unless otherwise stated, this data set is used)
- 100MB of traffic from the ISCX 2012 data set [12, 33], and
- 100MB of traffic from the BigFlows data set [3].

We also test against 100MB of randomly generated data. In all cases the algorithms read the data from a file in chunks of 25MB each and the cost of reading the data is included in the measurements.

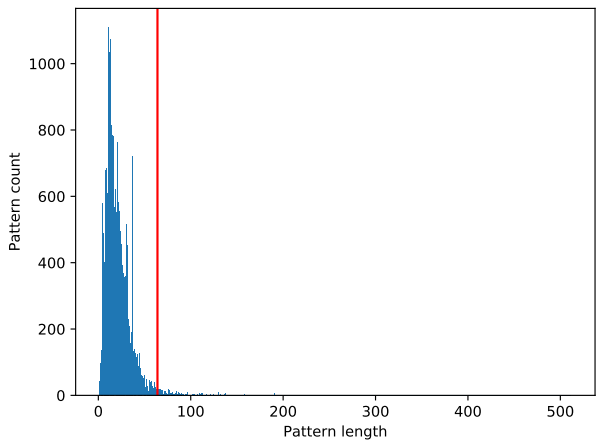
The set of malicious patterns to be matched are taken directly from Snort v2.9: we use a set of 2183 HTTP-related patterns from the default Snort distribution, as well as 5000 randomly chosen patterns from `emergingthreats.net`. Unless otherwise stated, the set of 2183 patterns is used for experiments.

Due to restrictions in OpenCL, statically allocating enough space to fit the longest possible pattern would be wasting memory resources. For this reason, we use no pattern longer than sixty-four (64) characters and remove any instance longer than this threshold. This choice is motivated by the distribution of pattern lengths in the first and second pattern data set, which is shown in Figures 5.2a and 5.2b respectively. In the case of the patterns from `emergingthreats.net`, the longest pattern is 513 characters, while most are much much shorter than that. Removing any pattern longer than 64 characters removes approximately 80 and 500 from the first and second pattern data set respectively. This decision to remove excessively long patterns is similar to what happens in systems such as

[1] We are aware of the artifacts in the DARPA 2000 set, and the discussions in the community about its suitability for measuring the *detection capability* of intrusion detection systems [23, 24]. In our experiments, we use it only for the purpose of comparing execution time and energy usage between algorithms, allowing for future comparisons on a known and easily-available dataset.



(a) Snort HTTP patterns



(b) All patterns from `emergingthreats.net`

Figure 5.2: Distribution of pattern lengths. Red line signifies 64 characters. Both pattern sets consist mostly of short patterns, less than 64 characters long.

Snort, where long patterns are truncated and only shorter versions of them are used in the pattern matching engine.

In order to ensure a fair comparison between the algorithms based on the considerations mentioned in Section 5.2, we have ensured that all versions have identical functionality, i.e., they all count the number of patterns that are matched in the input.

Finally, our main *performance criterion* used to compare the different versions is the *execution time* of the algorithm as a whole, which includes: reading data from a file, performing pattern matching, counting the number of matches and, for the GPU versions, mapping memory between the CPU and the GPU. We do not include the cost of pre-processing, e.g., building the state machines of PFAC or the filters of DFC, since this happens offline before deployment. When measuring energy consumption, we gather measurements, using the on-board energy sensors of the device, at a rate of 100Hz.

5.5.2 Deciding parameters for DFC

It is important to understand how the characteristics of the target platform affect the performance of the algorithms and allow for different optimizations. This analysis has been done already for PFAC by Aragon et al. [4], but not for DFC since we are the first to implement a GPU version of it. Therefore, we follow the approach of Aragon et al. and we present, in Table 5.3, the effects of the optimizations discussed in Section 5.4 on DFC's performance.

In order to limit the number of different configurations that need to be examined, we follow a greedy approach: we change one parameter until we find the value for which we get the best performance, i.e., the smallest TOTAL execution time, then keep that value and move on with the next parameter. The configurations and their effect in Table 5.3 are described below. In the results section of the table we report the time it takes to write/read data to/from the device (WRITE/READ), the execution time of the kernel (KERNEL) that implements pattern matching on the GPU, and the total measured time (TOTAL), as well as the energy consumed (ENERGY). Yellow boxes indicate configurations that improve the overall performance (TOTAL time) compared to the previous configuration, while red boxes indicate changes in configuration that do not have a positive effect on performance.

Reducing Memory transfers (MAP): The MAP configuration option is binary (yes/no) reflecting whether we use *map* to reduce memory copies. Overall, mapping memory instead of copying it has the most significant effect, seen from the reduction in the time it takes to read/write data to/from the device. This is expected since, as previously mentioned, the CPU and the GPU share the same

Configurations					Results				Improvement	
MAP	THR	MEM	VEC	WG	WRITE (ms)	READ (ms)	KERNEL (ms)	TOTAL (ms)	ENERGY (J)	EXE	ENE
NO	1	GLOB	NO	128	431	4870	2596	11170	2867	1	1
YES	1	GLOB	NO	128	198	945	2661	6787	1806	1.65	1.59
YES	8	GLOB	NO	128	195	678	1829	4118	991	2.71	2.89
YES	16	GLOB	NO	128	195	654	1599	3790	923	2.95	3.11
YES	24	GLOB	NO	128	196	653	1488	3670	875	3.04	3.28
YES	32	GLOB	NO	128	194	644	1427	3451	854	3.24	3.36
YES	40	GLOB	NO	128	196	648	1406	3440	845	3.25	3.39
YES	48	GLOB	NO	128	196	647	1412	3464	845	3.22	3.39
YES	40	LOC	NO	128	194	643	6267	8411	2238	1.33	1.28
YES	40	TEX	NO	128	197	646	1777	3897	996	2.87	2.88
YES	40	GLOB	YES	128	191	637	2065	4172	1095	2.68	2.62
YES	40	GLOB	NO	64	196	645	1674	3784	905	2.95	3.17
YES	40	GLOB	NO	256	196	645	1435	3501	870	3.19	3.29

Table 5.3: Summarized configuration impact for GPU version of DFC, similar to the evaluation methodology followed in [4]. Changes in configuration that result in reduction in execution time are marked with green, whereas changes that increase execution time are marked in red.

physical memory which makes memory copies redundant.

Increasing work per thread (THR): The THR configuration option controls the thread-granularity, i.e., how many characters each GPU thread processes. It is varied from 1 character per thread to 48. Increasing the thread granularity to more than one character per thread resulted in a great decrease in the kernel execution time, since there is now more work for each thread to do, instead of exiting early. However, at a certain point there are too few threads to keep the hardware pipeline busy and performance decreases.

Utilizing global, local or texture memory (MEM): The MEM configuration option has three cases, reflecting the type of memory where the filters are stored, i.e. global, local, or texture memory. As expected, using local memory to store the filters had a negative effect on performance, because the local memory in the Mali GPU is simulated by global memory. Moreover, this simulation seems to come at a cost in that the local memory is more expensive than the global memory with the same overall settings (which was also observed by Aragon et al. [4]). Surprisingly, using texture memory did not have a beneficial effect on performance either. This is likely due to the extra instructions needed to access and isolate the relevant bits from the filter when it is stored as a texture.

Vectorized execution (VEC): The VEC configuration option controls whether the kernel is vectorized. In related work [37] for the CPU, vectorization played a large role in improving the performance. However, vectorizing the kernel in our setting did not have a beneficial effect, likely due to the lack of proper *gather* operations, which means that we incur a penalty when switching between vectorized and scalar code. However, future architectural support for vectorized operations would likely bring improvements similar to what is seen in [37] for the CPU.

Altering OpenCL workgroup size (WG): Finally, the WG configuration option varies the workgroup size. The best configuration for the work-group size is 128 work-items per work-group.

5.5.3 Overall comparison

After establishing a set of parameters that works best for DFC (the best configurations from Section 5.5.2), in addition to the parameters that work best for PFAC from Aragon et al. [4], we put all algorithms to the test in this section. Figure 5.3 shows the execution time of all algorithms when processing the DARPA data set with the default set of 2183 patterns. We run each experiment five times and report the average execution time. In Figure 5.3 we have broken down the cost to its individual components. Post-processing refers to counting the number of matches, based on the results read from the GPU. Even though the main focus is

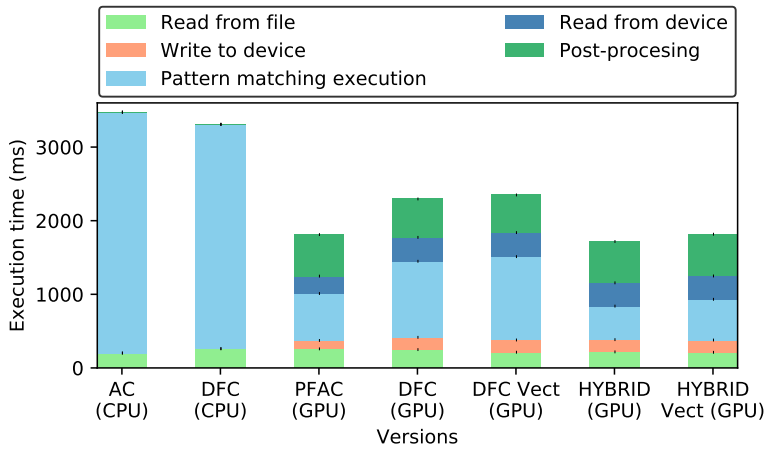


Figure 5.3: Execution time (broken down to its different components) of the different versions.

the execution time of pattern matching itself (light blue bars), we still present the additional costs for completeness.

When comparing results between the performance of the CPU baseline and the GPU, keep in mind that a direct comparison is not always straightforward. Here, we compare against the performance of parallel algorithms that have been transformed to operate on the GPU, against single-threaded algorithms that operate on the CPU. Using all the available threads in the CPU is likely to improve CPU performance, assuming there is an efficient way to parallelize the algorithms on the CPU and taking into account bottlenecks involved in CPU parallelization. Still, our comparison allows us to express the performance of the GPU in terms of something easily understood: the performance of a single CPU thread. Moreover, note that the GPU and CPU can complement each other: it is possible to have both the GPU and the CPU threads working simultaneously, either on disjoint parts of the input data, or on different tasks (e.g. the post-processing can be done in parallel by the CPU).

CPU vs GPU: First, comparing the CPU (the first two bars in Figure 5.3) and the GPU versions (the rest) shows that all GPU versions perform significantly better than the CPU versions, up to 2X less total execution time when comparing Aho-Corasick with *HYBRID*. That result supports the claim that embedded accelerators such as the Mali GPU can effectively offload pattern matching for Network Intrusion Detection applications, regardless of the family of algorithms used. Particularly, we notice that the execution cost of the pattern matching part

alone (light blue bars) is greatly reduced on the GPU by up to 7X, suggesting that pattern matching kernels make great use of the high degree of parallelism offered by the GPU. However, notice that the extra costs of data transfers and post-processing of the results are significant and offset, to some extent, the total benefit of offloading pattern matching on the GPU. Even though data copies are avoided, the overhead of mapping/unmapping the memory region every time the buffer is full is still significant in this application. As such, a new design that would minimize this effect further, e.g., by overlapping the CPU and the GPU execution, would be interesting to explore.

We now focus on the GPU versions and compare them with each other.

PFAC vs DFC (GPU): Comparing PFAC with DFC (GPU), we find the pattern matching part of PFAC is 1.62x faster than DFC (Figure 5.3). This is likely due to two reasons. First, the cost of verification in DFC is high, because it includes accesses to hash tables containing the patterns as well as exact matching. Second, DFC was originally designed to have an increased instruction count compared to Aho-Corasick (more instructions needed to access filters and hash tables) but a better cache utilization. In GPUs, the benefit from high cache utilization is not as important as in CPUs, due to the large number of threads spawned and the device's ability to switch between threads quickly and hide the high memory latency incurred when there is a cache-miss. As a result, the benefits of cache utilization are offset from the cost of the extra instructions and the costly verification phase.

Interestingly, the vectorized version of DFC (DFC Vect (GPU)) fails to bring any speedup. As mentioned in Section 5.5.2, vectorizing the kernel for each GPU thread is not efficient for this workload, due to the lack of *gather* operations.

The *HYBRID* approach: Among the GPU versions, *HYBRID* manages to execute the pattern matching part faster than both PFAC and DFC (1.3X faster than PFAC and 2.2X faster than DFC). Contrary to DFC, *HYBRID* avoids the costly verification by switching to PFAC when there is a hit in the filters, while still keeping the benefits of using filters to quickly filter out parts of the input that cannot contain a match. As in DFC though, the vectorized version (*HYBRID* Vect (GPU)) does not bring a speedup.

Energy Comparison: Finally, in Figure 5.4 we report the total consumed energy for the same experiment, across the different versions. Overall, we see that the reduction in execution time from the previous figure translates, in almost all cases, directly to reduction in consumed energy, with the best GPU version (*HYBRID*) consuming 2X less energy than Aho-Corasick. In this figure we also show separately the energy consumed by the CPU (A15) and the GPU. Notice that the GPU versions still consume significant CPU energy, mostly due to post-processing and idly waiting for the GPU execution to finish.

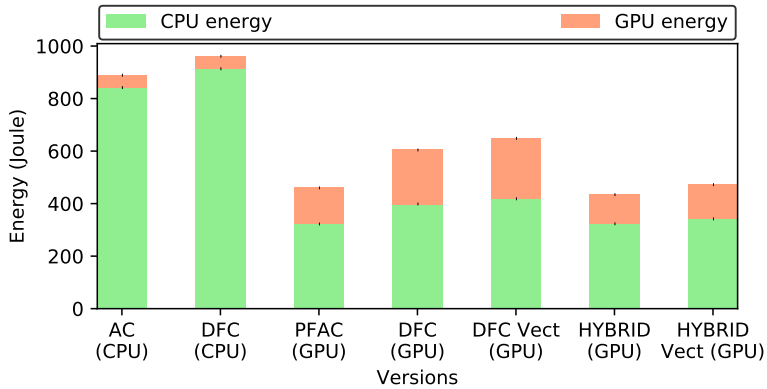


Figure 5.4: The CPU and GPU energy consumed by the different versions.

5.5.4 Varying the data sets and the number of patterns

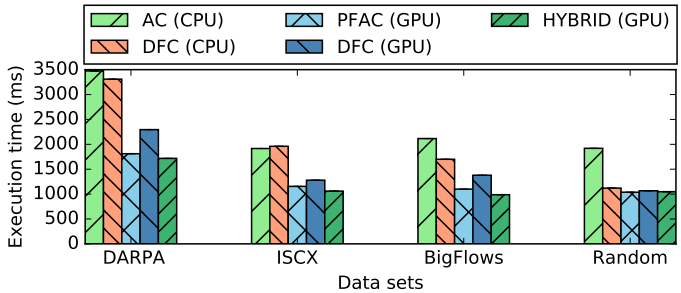
In this section, we evaluate the behavior of the different algorithms across different data sets and number of patterns. Insights about this behavior are important when considering real packet processing deployments where the characteristics of the incoming traffic might change or new malicious patterns might be added in the database.

Varying the data sets: Figure 5.5a shows the overall cost (including reading from file, data transfers etc.) for different real and synthetic data sets, when using the default set of patterns. Here, we have omitted the vectorized versions since they do not bring a significant change in performance. In all cases, the GPU versions outperform the CPU, with *HYBRID* having the smallest total execution time in almost all cases.

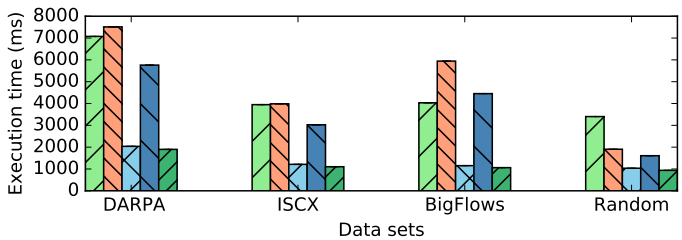
An interesting observation is that, when using randomly generated data, the CPU version of DFC performs significantly better than with real traffic. This is because randomly generated characters are very unlikely to cause hits in the filters, reducing the need for verification and most of the memory accesses are now served by the CPU cache [10]. This stresses the need to use realistic data sets when comparing algorithms in a benchmark.

Increasing the number of patterns: In Figure 5.5b, we increase the number of malicious patterns to 5000. Overall, we see that the execution time of all versions increases (notice the different range on the y-axis between Figures 5.5a and 5.5b). Aho-Corasick nearly doubles in execution time, mostly due to the fact that the state machine grows and does not fit the CPU cache.

Interestingly, GPU versions that also use a state machine, i.e., PFAC and



(a) 2183 patterns



(b) 5000 patterns

Figure 5.5: Execution time across different data sets when using a) the default set of 2183 patterns and b) 5000 randomly chosen patterns.

HYBRID, do not get affected to such a great extent. This is, again, due to the fact that GPU cache misses are not detrimental if the GPU is able to hide the latency of accessing the main memory, by switching between many active threads. We also notice that both DFC versions increase significantly in execution time, likely due to the exact matching part of the verification step of DFC, which is costly and happens more often when the number of patterns increases.

5.5.5 Deciding a filter size for *HYBRID*

In this section, we take a closer look at the *HYBRID* approach and specifically at the effect of the filter size on the performance of the algorithm. Intuitively, a larger filter will be more sparsely populated, meaning that we expect fewer hits (therefore fewer times that we need to resort to the state machine of PFAC) when filtering the input. On the other hand, a small filter can fit in the GPU cache

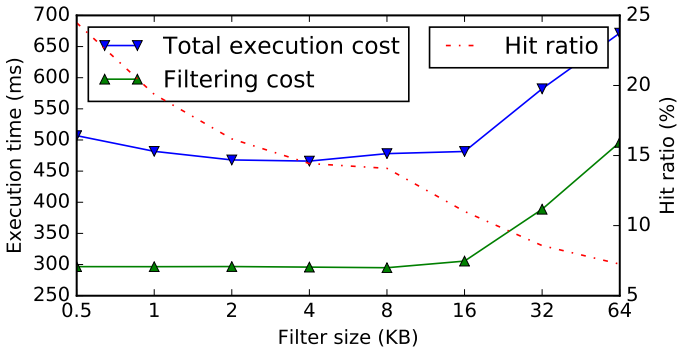


Figure 5.6: Cost of filtering and total GPU execution time of the *HYBRID* approach (left y-axis), as well as the effectiveness of the filtering (hit ratio, right y-axis), as we increase the filter size.

(16KB per shader core), making it easier to access.

In the following experiment, we use the first three characters from each pattern to populate the filter, after hashing them with a simple multiplicative hash function. We vary the effective size of the filter by masking the hash value appropriately, effectively bounding the size of the filter. In Figure 5.6 we report: (i) the cost of only accessing the filter (green line, left y-axis) (ii) the hit ratio of the filter, i.e., the number of hits in the filter compared to the total number of input characters (red line, right y-axis) and (iii) the total cost of the *HYBRID* execution on the GPU, i.e., the cost of both filtering and traversing the state machine of PFAC when there is a hit in the filter (blue line, left y-axis).

As expected, the hit ratio decreases as the filter becomes larger, ranging from 25% hit rate for a half-KB filter to 5% hit rate for a 64KB filter. The cost of accessing the filter remains small while the filter is smaller than the size of the cache (16KB) and increases rapidly afterwards. The best performing configuration is when the filter size is much less than the size of the cache, because we still need to save space for the state machine for when there is a match.

5.5.6 Summary of the results

In this section, we presented the results of our benchmark where multiple pattern matching algorithms are brought to the test on a medium/high range IoT device with an embedded GPU. Experiments using real data sets and patterns showed that: (i) the GPU is a viable alternative for pattern matching on these devices, both

in terms of execution time and energy consumption and (ii) there were significant differences in performance between the GPU based algorithms, uncovering the strength and weaknesses of each approach. Stemming from this analysis, it was possible to identify new meaningful combinations (*HYBRID*) that combine techniques from existing work and outperform them. The co-evaluation of CPU and GPU algorithms also uncovered how different algorithms utilize the hardware's resources differently: in the CPU, having good cache locality proved important, whereas it mattered less in the GPU. Finally, using both synthetic and real data sets showed that most pattern matching algorithms are highly data-dependent, raising interesting future directions about the feasibility to adapt to the distribution of the data.

5.6 Related work

Pattern matching has been an active field of research for decades and the literature offers numerous algorithms for a variety of different settings. On single string pattern matching, Boyer-Moore [9] and Knuth-Morris-Pratt [18] are two well-known algorithms that skip over parts of the input and perform pattern matching in sub-linear time. However, such algorithms do not work well in the context of pattern matching for intrusion detection where there are many patterns to search for simultaneously. An important multiple string matching algorithm, other than the Aho-Corasick [1] we already summarised in Section 5.3, is the Wu-Manber [43] algorithm that keeps a table to store information on how many bytes we can skip from the input. Hyperscan [41] is an open source regular-expression library that also includes many optimizations for fixed string pattern matching. However, these optimizations are built around Intel's high-end vector instruction set extensions and are not available in the ARM-based platform we use in this work.

Apart from the pattern matching algorithms included in this work (and summarized in Section 5.3), there are others that target GPU platforms. FFBF [26] is a filter based approach by Moraru and Andersen that uses Bloom filters to find a subset of the input and the patterns that should be matched together. However, FFBF imposes restrictions on the pattern size and requires long patterns in order to work effectively. On the contrary, the approaches we consider are flexible with respect to the number of patterns. Kouzinopoulos et al. [19] also experiment with pattern matching algorithms on GPUs, using the CUDA framework. In [8], Bellekens et al. present a compressed Aho-Corasick algorithm that improves the bandwidth of data transfers on both NIDS and DNA sequencing workloads. All of the above-mentioned work targets high-end, desktop GPUs. In this work,

we focus on embedded GPUs that have a significantly different architecture, as described in Section 5.4.

On the topic of embedded GPUs, as already mentioned, Aragon et al. [4] implement PFAC in OpenCL and report its performance on two embedded GPUs that have almost the same architecture as the one used in this paper. In [22], Maghazeh et al. benchmark various GPGPU applications on an embedded GPU, concluding that the high energy efficiency of such devices makes them a promising choice for a wide range of workloads such as genetic algorithms and vector similarity (referred to as pattern matching in their work). In [14], Grasso et al. present optimizations techniques for the Mali GPU that allow them to gain significant speedups (both in terms of execution time, as well as energy) for various benchmarks.

Even though this work focuses on the pattern matching algorithms themselves, it is important to mention work that considers the role of GPUs in the NIDS as a whole. Vasiliadis et al. [38] use Aho-Corasick to build a GPU based NIDS. In [39], they also integrate more network processing workloads into a general GPU framework, including flow state management and TCP stream reconstruction. Go et al. [13] experiment with integrated GPUs and show that they provide an appealing alternative for packet processing workloads, including pattern matching. There is also work that considers both the CPU and the GPU and how they can coordinate to better serve the needs of the NIDS. Kim et al. [17] propose NBA, a framework that abstracts the GPU offloading from the programmer and includes load balancing and batching. Their NIDS implementation is also based on Aho-Corasick. Vasiliadis et al. [40] present a system based on Snort that uses all CPU threads and multiple GPU devices. Jamshed et al. [15] present Kargus, a similar parallel design that is based on their own custom NIDS. Papadogiannaki et al. [30] extend the existing work in the field with a scheduler that decides the placement of the different computing tasks of the NIDS, across a heterogeneous platform.

5.7 Conclusions

In this paper, we introduce a fair and thorough co-evaluation of pattern matching algorithms for network intrusion detection on embedded devices with GPUs with the aim to methodologically investigate the algorithm behavior in conjunction with the architectural features available on medium to high-level devices used in deployments for the Internet of Things. We present results from existing approaches, in-house implementations of state of the art algorithms, as well as a new algorithm, *HYBRID*, that combines the benefits from existing designs.

We conclude that GPUs on embedded devices are an attractive alternative to CPUs when it comes to pattern matching for intrusion detection, since the GPU-based algorithms in our benchmarks managed to reduce the overall execution time and energy consumption of pattern matching workloads by up to 2 times. We investigate how algorithm engineering approaches that are based on the platform's architectural features, e.g., shared GPU and CPU memory, affect the performance of various algorithms. Finally, we show that *HYBRID* is a good fit for the GPU, achieving up to 1.4x speedup compared to the best GPU-based baseline, suggesting that hardware awareness in the design of security applications leads to significant performance improvements. This investigation provides a baseline for the community to further develop algorithms and make standard security tool deployable in IoT devices. Implementations used in the benchmark are available online[2].

Acknowledgements

The research leading to these results has been partially supported by the Swedish Civil Contingencies Agency (MSB) through the projects RICS and RIOT, by the Swedish Foundation for Strategic Research (SSF) through the framework project FiC, by the Swedish Research Council (VR) through the project ChaosNet and the project AgreeOnIT, the Vinnova-funded project "KIDSAM", and from the European Community's Horizon 2020 Framework Programme under grant agreement 773717.

Bibliography

[1] Alfred V. Aho and Margaret J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM*, 18(6):333–340, June 1975.

[2] Spyros Antonatos, Kostas G. Anagnostakis, and Evangelos P. Markatos. Generating realistic workloads for network intrusion detection systems. *SIGSOFT Softw. Eng. Notes*, 29, 2004.

[3] Appneta. Sample captures. `http://tcpreplay.appneta.com/wiki/captures.html/` [Accessed: 2018-09-18].

[4] Elena Aragon, Juan M. Jiménez, Arian Maghazeh, Jim Rasmusson, and Unmesh D. Bordoloi. Pattern matching in opencl: Gpu vs cpu energy consumption on two

[2] `https://bitbucket.org/mpastyl/acsac_pattern_matching_benchmark_opencl/src/master/`

mobile chipsets. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*, IWOCL '14, pages 5:1–5:7, New York, NY, USA, 2014. ACM.

[5] ARM. ARM Mali-T628 product page. `https://www.arm.com/products/multimedia/mali-cost-efficient-graphics/mali-t628.php`. Accessed: 2018-03-14.

[6] Arm. ODROID-XU3. `https://developer.arm.com/graphics/development-platforms/odroid-xu3`. Accessed: 2018-05-25.

[7] ARM. ARM Mali GPU OpenCL, Version 3.0, Developer Guide. `https://static.docs.arm.com/100614/0300/arm_mali_gpu_opencl_developer_guide_100614_0300_00_en.pdf`, 2018. Accessed: 2018-03-14.

[8] Xavier JA Bellekens, Christos Tachtatzis, Robert C Atkinson, Craig Renfrew, and Tony Kirkham. A highly-efficient memory-compression scheme for gpu-accelerated intrusion detection systems. In *Proceedings of the 7th International Conference on Security of Information and Networks*, page 302. ACM, 2014.

[9] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, October 1977.

[10] Byungkwon Choi, Jongwook Chae, Muhammad Jamshed, KyoungSoo Park, and Dongsu Han. DFC: Accelerating string pattern matching for network applications. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 551–565, Santa Clara, CA, 2016. USENIX Association.

[11] Cisco. Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are. White Paper `https://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf`, 2015. Accessed: 2018-05-07.

[12] Canadian Institute for Cybersecurity. UNB ISCX intrusion detection evaluation dataset. `http://www.unb.ca/research/iscx/dataset/iscx-IDS-dataset.html`, 2012. Accessed: 2016-12-10.

[13] Younghwan Go, Muhammad Asim Jamshed, YoungGyoun Moon, Changho Hwang, and KyoungSoo Park. APUNet: Revitalizing GPU as Packet Processing Accelerator. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 83–96, Boston, MA, 2017. USENIX Association.

[14] Ivan Grasso, Petar Radojkovic, Nikola Rajovic, Isaac Gelado, and Alex Ramirez. Energy efficient hpc on embedded socs: Optimization techniques for mali gpu. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 123–132, May 2014.

[15] Muhammad Asim Jamshed, Jihyung Lee, Sangwoo Moon, Insu Yun, Deokjin Kim, Sungryoul Lee, Yung Yi, and KyoungSoo Park. Kargus: A Highly-scalable Software-based Intrusion Detection System. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 317–328, New York, NY, USA, 2012. ACM.

[16] Khronos Group. OpenCL Overview. https://www.khronos.org/opencl/. Accessed: 2018-03-11.

[17] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. Nba (network balancing act): A high-performance packet processing framework for heterogeneous processors. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 22:1–22:14, New York, NY, USA, 2015. ACM.

[18] Donald Knuth, James Morris, Jr., and Vaughan Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[19] Charalampos S Kouzinopoulos and Konstantinos G Margaritis. String matching on a multicore GPU using CUDA. In *Informatics, PCI'09. 13th Panhellenic Con. on*. IEEE, 2009.

[20] Lincoln Laboratory. DARPA intrusion detection data sets. https://www.ll.mit.edu/r-d/datasets/2000-darpa-intrusion-detection-scenario-specific-data-sets, 2000. Accessed: 2018-09-20.

[21] Cheng-Hung Lin, Chen-Hsiung Liu, Lung-Sheng Chien, and Shih-Chieh Chang. Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs. *IEEE Transactions on Computers*, 62(10):1906–1916, Oct 2013.

[22] Arian Maghazeh, Unmesh D. Bordoloi, Petru Eles, and Zebo Peng. General purpose computing on low-power embedded gpus: Has it come of age? In *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 1–10, July 2013.

[23] Matthew V Mahoney and Philip K Chan. An analysis of the 1999 DARPA/Lincoln Laboratory evaluation data for network anomaly detection. In *Int. Workshop on Recent Advances in Intrusion Detection*. Springer, 2003.

[24] John McHugh. Testing intrusion detection systems: a critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by lincoln laboratory. *ACM Tran. on Information and System Security (TISSEC)*, 3(4):262–294, 2000.

[25] MichaelS. Gather Scatter operations. http://insidehpc.com/2015/05/gather-scatter-operations/, 2015. Accessed: 2016-12-10.

[26] Iulian Moraru and David G. Andersen. Exact Pattern Matching with Feed-forward Bloom Filters. *J. Exp. Algorithmics*, 17:3.4:3.1–3.4:3.18, September 2012.

[27] Marc Norton. White paper: Optimizing pattern matching for intrusion detection. Technical report, Snort, 2004.

[28] Nvidia. About CUDA. https://developer.nvidia.com/about-cuda. Accessed: 2018-03-11.

[29] NVIDIA. Jetson Nano Brings AI Computing to Everyone. https://devblogs.nvidia.com/jetson-nano-ai-computing/. Accessed: 2019-04-17.

[30] Eva Papadogiannaki, Lazaros Koromilas, Giorgos Vasiliadis, and Sotiris Ioannidis. Efficient software packet processing on heterogeneous and asymmetric hardware architectures. *IEEE/ACM Transactions on Networking*, 25(3):1593–1606, June 2017.

[31] David E. Sanger and Nicole Perlroth. A New Era of Internet Attacks Powered by Everyday Devices. `https://nytimes.com/2016/10/23/us/politics/a-new-era-of-internet-attacks-powered-by-everyday-devices.html`, 2016. Accessed: 2018-03-04.

[32] Alessandro Sforzin, Félix Gómez Mármol, Mauro Conti, and Jens-Matthias Bohli. RPiDS: Raspberry Pi IDS - A Fruitful Intrusion Detection System for IoT. In *UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld, Toulouse, France, July 18-21, 2016*, pages 440–448, 2016.

[33] Ali Shiravi, Hadi Shiravi, Mahbod Tavallaee, and Ali A. Ghorbani. Toward developing a systematic approach to generate benchmark datasets for intrusion detection. *Computers & Security*, 31(3), 2012.

[34] Snort. Snort++. `https://github.com/snort3/snort3`. Accessed: 2018-12-21.

[35] Snort Network Intrusion Detection and Prevention System. `https://www.snort.org`. Accessed: 2018-09-21.

[36] Ioannis Sourdis and Dionisios Pnevmatikatos. Fast, large-scale string match for a 10gbps fpga-based network intrusion detection system. In Peter Y. K. Cheung and George A. Constantinides, editors, *Field Programmable Logic and Application*, pages 880–889, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[37] Charalampos Stylianopoulos, Magnus Almgren, Olaf Landsiedel, and Marina Papatriantafilou. Multiple pattern matching for network security applications: Acceleration through vectorization. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 472–482, Aug 2017.

[38] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In Richard Lippmann, Engin Kirda, and Ari Trachtenberg, editors, *Recent Advances in Intrusion Detection*, pages 116–134, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[39] Giorgos Vasiliadis, Lazaros Koromilas, Michalis Polychronakis, and Sotiris Ioannidis. GASPP: A gpu-accelerated stateful packet processing framework. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 321–332, Philadelphia, PA, 2014. USENIX Association.

[40] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. Midea: A multi-parallel intrusion detection architecture. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, New York, NY, USA, 2011. ACM.

[41] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. Hyperscan: A fast multi-pattern regex matcher for modern cpus. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 631–648, Boston, MA, 2019. USENIX Association.

[42] Wang Wei. Casino gets hacked through its internet-connected fish tank thermometer. `https://thehackernews.com/2018/04/iot-hacking-thermometer.html`, 2018. Accessed: 2019-01-16.

[43] Sun Wu and Udi Manber. A fast algorithm for multi-pattern searching. Technical Report TR-94-17, University of Arizona. Department of Computer Science, 1994.

Part IV

Distributed Processing on Resource-Constrained Devices

PAPER V

Charalampos Stylianopoulos, Magnus Almgren,
Olaf Landsiedel, Marina Papatriantafilou

Geometric Monitoring in Action: a Systems Perspective for the Internet of Things

Adapted and expanded version of the paper that appeared in *the Proceedings of the IEEE 43rd Conference on Local Computer Networks (LCN)*, pp. 433-436, IEEE 2018.

6

Geometric Monitoring in Action: a Systems Perspective for the Internet of Things

Abstract

Many applications in the Internet of Things (IoT) continuously monitor sensor values and react when their network-wide aggregate exceeds a threshold. Geometric monitoring (GM) is a methodology that promises a several-fold reduction in terms of communication and coordination between the sensors for such applications. Previous work on GM has been limited to analytic or high-level simulation results and does not consider critical system aspects such as the communication protocol, involving parameters such as the radio duty-cycle and packet losses.

In this paper, we devise, realize and evaluate a system design for GM, enabling deployment possibilities on resource-constrained IoT devices and network stacks. In particular we provide (i) an algorithmic implementation for commodity IoT hardware (ii) a study and insights regarding duty cycle reduction and energy savings on actual IoT nodes, in connection with existing analytic and high-level simulation results about GM, as well as (iii) a study and insights on the influence of packet losses on the effectiveness of the method, in terms of accuracy and responsiveness. Our results, both from full-system simulations and a publicly available testbed, show that GM indeed provides several-fold energy savings in communication; e.g., we see up to 3x and 11x reduction in duty-cycle when monitoring the variance and average temperature of a real-world data set, respectively but the results fall short of the analytic predictions on the expected savings in the number of exchanged messages (4.3x and 44x, respectively). Hence, we investigate the energy overhead imposed by the network stack as well as the effects of packet losses on the accuracy and responsiveness of the algorithm. Through the above insights, we offer guidelines for the adaption of GM and similar algorithms in IoT settings.

6.1 Introduction

Monitoring system-state or the environment-conditions are fundamental uses for Wireless Sensor Networks (WSNs). Given a set of sensor nodes $n_1, ..., n_N$ with readings $\vec{v_1}, ..., \vec{v_N}$ that vary over time, we want to continuously track whether a function f, defined over the network-wide weighted average of the readings, is larger than a threshold T. Keeping track of such a function serves as a basis for many applications, e.g. detecting outliers [2], hot-spots [25] or denial-of-service attacks [7] [8, 12]. The challenge is to let all nodes accurately determine whether the function is above or below the threshold locally, without having to share every reading with the others. For simple, linear functions (average), local

constraints can be derived to minimize communication, while for non-linear functions (variance), deriving such constraints is challenging.

Distributed monitoring has received high interest, with many proposed solutions. Sharfman et al. [24] proposed a general method, called *geometric monitoring* (GM), that can monitor *any* function, linear or not, computed over network-wide aggregates and can keep track of its value with respect to a threshold. The method suppresses unnecessary communication by deriving local constraints that individual nodes can check without communication. The effectiveness of the method has been thoroughly studied showing impressive communication reduction results. Variations of GM have been enhanced with sketches [10] and prediction models [11] and have been applied on outlier detection [2] and data stream queries [9]. The above mentioned extensions are orthogonal to the original GM algorithm. In this paper we focus on the basic principles of GM and tackle the challenges described next.

Research Challenges: Even though GM and similar threshold monitoring algorithms are designed with sensor networks in mind, there is lack of insights from full-system perspective. Existing work on GM has focused on the algorithmic part, backed up with numerical, high level simulations where communication is assumed instant and reliable. However, the reality is different: packet losses are frequent, nodes have severe constraints on processing power and lifetime, and message propagation is costly, both in terms of energy as well as latency. Recent work on data aggregation [23] has shown that properties such as the radio duty cycle greatly influence the lifetime savings that can be achieved in practice.

Thus, the feasibility of GM for WSNs and the impact of the system's properties raise questions, such as: (i) What are the actual battery lifetime savings that we can achieve on real nodes? (ii) What is the effect of packet losses on the accuracy and responsiveness of the algorithm? (iii) How can such methods be implemented on commodity IoT hardware?

To address the aforementioned questions, we take a step beyond the existing analysis and consider the whole system stack, through (i) extensive, cycle-accurate, full-system simulations and (ii) validation from a real deployment. We are thus able to evaluate up to what degree and condition, emerging results on distributed continuous monitoring can benefit real WSN deployments.

Contributions:

- We bridge the gap between high level numerical simulation results on threshold monitoring and real IoT environments. We study the algorithmic implementation and the actual performance on a real deployment, using real data sets and offer new insights.

- As the computational complexity of GM is a serious challenge for CPU-

constrained devices, we suggest an efficient approximation.[1]

- We show that the practical energy lifetime improvements may vary significantly and are often far from the savings estimated analytically. Specifically, we find that the overhead of idle listening is a dominant factor that can sometimes limit the effectiveness of GM.
- Communication patterns under GM vary greatly over time, with periods of no activity and bursts of concurrent updates. This presents a challenge to the underlying networking protocol.
- We study message losses and identify that they cause the nodes to desynchronize, with cascading effects, reducing the accuracy and responsiveness of the algorithm.

The remainder of this paper is organized as follows. Section 6.2 summarizes GM and introduces its challenges in wireless sensor networks. Section 6.3 outlines system design challenges and our solutions as well as tunable parameters of the network stack that will influence properties of the system implementation. We evaluate our full-system implementation of GM in Section 6.5, through the experimental methodology presented in Section 6.4. We discuss related work in Section 6.6 and conclude in Section 6.7.

6.2 Overview of the problem

Here we summarize GM as a general method for distributed threshold monitoring. We then outline background related to wireless sensor networks and the system model.

6.2.1 The Geometric Monitoring Method (GM)

In their seminal work [24], Sharfman et al. present a general distributed method able to monitor (with respect to a threshold) arbitrary functions defined over network-wide aggregates. Instead of having nodes broadcast at every *epoch* (sensor sampling period), each node uses only information from the last available

[1] A short version of the paper presented in this chapter was published in the proceedings of the IEEE 43rd Conference on Local Computer Networks (LCN). In this chapter, we include the extended version of the paper. Due to that, there is some overlap between this chapter and Chapter 7 when it comes to the approximation method that reduced the computational complexity of GM.

broadcast and its current (up-to-date) sensor reading, to calculate a region of the input domain where the true value of the network-wide aggregate can be. As long as this region remains fully on one side of the threshold, communication is avoided; Otherwise, broadcasts are needed to see whether the local change is offset by changes at other nodes (false alarm) or if the function has actually crossed the threshold. The key points of the method are briefly explained here, but the interested reader is referred to [24] for a full explanation with proofs.

In GM, the set of sensor readings $\vec{v_1}, ..., \vec{v_N}$ are called *local statistics vectors*. These vectors are only known locally, but sporadically a node n_i will broadcast its $\vec{v_i}$ to every other node. The last broadcasted value from n_i is denoted as $\vec{v_i'}$. The weighted average[2] of the local vectors (eq 6.1) is called the *global statistics vector*.

$$\vec{v} = \sum_{i=1}^{N} w_i * \vec{v_i} \tag{6.1}$$

Similarly, the weighted average of the last broadcasted values is called the *estimate vector* ($\vec{e}$); it is known to all nodes and is an estimate of the global statistics vector.

When a node measures a new set of sensor readings, its local statistics vector will drift ($\vec{\Delta v_i} = \vec{v_i} - \vec{v_i'}$). The *drift vector* $\vec{u_i}$ is defined as the displacement of the estimate vector because of the new drift, i.e. $\vec{u_i} = \vec{e} + \vec{\Delta v_i}$ and can be computed locally without communication. Figure 6.1 shows an example of the method, also depicting the values defined above.

The convex hull of the drift vectors (yellow area) is defined as the set of all the convex combinations of $\vec{u_i}$ ($\sum \theta_i \vec{u_i}$).[3] As such, it is clear that the weighted average of the drift vectors (defined similar to eq 6.1) would be part of this set. With simple substitution, one can also see that the global estimate is equal to the weighted average of the drift vectors and thus it must also lie in the convex hull of the drift vectors. Thus, as long as the convex hull does not cross the threshold (i.e. lies in the white area), also $\vec{v}$ is guaranteed to not have crossed the threshold. However, nodes cannot locally determine the convex hull, as that would require knowing all $\vec{u_1}, ..., \vec{u_N}$.

This is where the final part of the method comes into play. Let each node create a sphere locally, centered at $\frac{\vec{e}+\vec{u_i}}{2}$ with a radius of $\frac{\vec{e}-\vec{u_i}}{2}$. This is possible since $\vec{u_i}$ is known to n_i and $\vec{e}$ is the same across all nodes at a given time. Sharfman et al. [24] prove that the union of those spheres strictly covers the

[2] W.l.o.g., we assume $0 \leq w_i \leq 1$ and that the weights sum to one.

[3] `E.g.https://www.maa.org/sites/default/files/pdf/upload_library/22/Ford/VictorKlee.pdf`

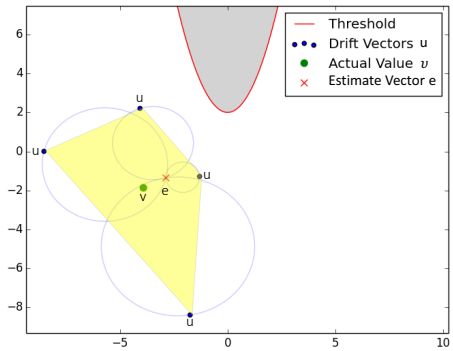


Figure 6.1: An example illustrating the GM method. The method keeps track of whether the actual value (green dot) is over or under the threshold (red line), by monitoring the circles formed by the drift vectors (gray circles).

convex hull. Therefore, a node only needs to track whether its locally computed sphere crosses the threshold. If yes, it will send its local vector to everyone, subsequently updating the estimate vector; else, it can remain quiet.

6.2.2 In the context of wireless sensor networks (WSNs)

Deploying an algorithm over WSNs can be challenging. WSNs are multi-hop, as the radios have a limited range. The communication is commonly over noisy, unreliable links. As such, the performance of an algorithm can be influenced by parameters and network properties, in ways that are hard to account for in an analytic study.

Regarding energy, the radio is the most consuming component. Consequently, the goal of most communication protocols is *radio duty cycling* (RDC), i.e. ensuring the radio is kept off as much as possible. In its simplest form, the RDC layer ensures that nodes turn their radio on a fixed number of times per second (called the *channel check rate* (CCR), aka wake-up time). On transmission, nodes keep transmitting the same packet for at least $1/CCR$ seconds, as in that time all neighbors have listened. High CCR suits frequent communication: broadcasts are shorter and nodes wake up more often to receive them; low CCR might be a better choice when communication happens rarely, so nodes do not have to check the medium often.

6.3 Applied GM and algorithmic implementation on Wireless IoT Sensors

For GM-based continuous threshold monitoring in IoT environments, we argue that focal points are: (i) Design challenges from the application's point of view and (ii) System properties and parameters affecting the design.

6.3.1 Addressing system challenges: processing and communication

Multi-hop, all-to-all communication:

In traditional WSN communication scenarios, either all nodes send to a single, fixed node (data collection) or all traffic is disseminated from a single, fixed node to all the others (data dissemination). With GM's communication requirements, every node can potentially be a source of information that needs to be disseminated to all other nodes (all-to-all communication) and even concurrently with other nodes (as shown in Section 6.5). In addition, sensor networks are commonly multi-hop, so an individual update generated by a single node needs to be propagated in a reliable manner to all nodes.

We consider mesh, unstructured networks that follow a simple approach for multi-hop propagation: every node that receives a packet with new information, will broadcast it further on. Obviously, this leads to an increased amount of broadcasts for every update. This is a commonly considered baseline, motivated by its inherent property that the update will eventually propagate throughout the network with a high degree of reliability, without the need to maintain a routing topology. As the goal of GM is to reduce the number of updates that need to be propagated, we expect that network-wide flooding of updates will not happen often. We evaluate this further in Section 6.5.

Recovery from losses.

The related literature on GM and similar methods does not typically consider packet losses but assumes reliable message delivery. In WSNs, losses are common and an important consideration for application design.

If an update from node A fails to reach B, node B will have stale information about A and the estimate vector will be *out of sync* (with respect to A), until the next update from A. An out of sync node has an inaccurate view of the network-wide aggregate being monitored and might miss a threshold violation or report a non-existing one.

We allow updates to get lost and rely on the application layer to eventually converge to the correct estimate vector. We evaluate the effect of losses on the *accuracy* of the algorithm and the time that a node might be out of sync in Section 6.5.

Threshold checking complexity.

As explained in Section 6.2, GM relies on computing a sphere, based on the estimate vector and the local drift and checking whether that sphere crosses a threshold surface. This can be computationally challenging, even in low dimensions, considering that the threshold surface might have an arbitrary shape. Even if there is a closed-form solution that accurately solves this problem, it can be computationally expensive. This calculation needs to be performed for every new sampled local reading and every new received update. The problem is particularly hard for sensor nodes, where the computational resources are scarce: there is often not any native-floating point support and the nodes need to go to sleep as soon as possible, to save energy.

We address this by approximating the GM-spheres (Section 6.2) with a simpler shape that makes the computations significantly simpler, while ensuring that we do not introduce false negatives. As an example in 2D, the spheres (circles for the 2D case) that are part of the GM method can be replaced with squares containing the former, which results in simpler boundary conditions (as it is simpler to check whether the sides of a square, rather than points on a circle, cross a surface). Obviously, there will be cases where the square check will report a violation even though the circles do not actually cross the threshold, hence sacrificing accuracy for communication reduction. However, a similar trade-off is already inherent in GM as there will be cases when spheres cross the threshold even though the full convex hull (cf. Section 6.2) would not. This relaxation might not cope with high dimensions, but in many cases of monitoring statistics such as variance or correlations between nodes, it provides a simple and efficient solution. In Section 6.6 we discuss other alternative, shape-sensitive extensions to GM that reduce the computational cost of threshold checking by approximating the threshold surface. Contrary to those methods, we simplify the threshold checking with an approximation that is particularly useful for the important case of tracking two variables (e.g. when monitoring the variance, the computation time decreases from 20ms to just 2ms).

Experiment label	Duration per exper.	Platform	Parameters	Metrics of interest
A: Full system simulations	20 hours	Cooja	CCR	DC, Comm. reduction, Lifetime impr., Loss rate, Latency
B: Testbed validation exper.	2 x 3 hours	Flocklab	Data set section	DC, Comm. reduction, Lifetime impr., Loss rate, Latency
C: Runtime insights	20 hours	Cooja	Elapsed time	DC, Number of updates
D: Accuracy & Responsiveness	10 hours / 1 hour	Cooja	Artificial loss rate	Average time out of sync, Responsiveness

Table 6.1: Summary of the experiments presented in this section.

6.3.2 Tunable system-parameters

Since a node might, at any time, propagate an update, we use a network stack based on asynchronous transmissions and RDC to save energy. In this setting, the main parameter of interest is the channel check rate (CCR).

As mentioned before, CCR affects: (i) how often nodes wake up to check the medium for possible transmissions and (ii) for how long a broadcasting node should keep re-transmitting a packet to make sure that all nodes receive it.

As the main goal of GM is to reduce the number of inter-node updates, one would generally expect GM to benefit from lower CCR values, compared to a naive approach that shares every sensor reading. On the other hand, GM's communication behaviour is highly data-dependent and unpredictable. There are periods with high activity and low activity, depending on the value that is being monitored and how close it is to the threshold. In addition, when an update is received by a node, the resulting recalculated global estimate ($\vec{u}$) might also cause a violation, forcing the node to broadcast its readings immediately, creating periods of burst traffic.

Based on the above, it is clear that: (i) CCR is a system parameter that will affect the expected energy savings of the method, and (ii) it is hard to come up with a value for CCR that can match the communication of GM at all times. We evaluate this further in Section 6.5.

6.4 Experimental methodology

We implemented geometric monitoring in Contiki [5], a well-known operating system for IoT applications. We targeted the TelosB platform and used a mainstream stack that relies on ContikiMac [4] for RDC. In this section, we assess the performance of GM in practice over a real network stack.

Experiment Setup: We run our experiments in two settings: (i) A *full-system evaluation* on the Flocklab [18] testbed, a deployment of TelosB nodes on a university building with 26 nodes on a four hop topology. Flocklab has realistic interference due to the presence of people and Wi-Fi signals and allows us to evaluate the system in a realistic environment. (ii) A *full-system simulation* on Cooja [22], a cycle-accurate simulator where the *whole network stack* is simulated in software at every node. Cooja simulates a real deployment as accurately as possible and we use it to run long experiments (up to 20 hours per configuration) that would not be possible in the testbed due to usage restrictions. The simulation platform also allows for repeatable experiments and fine-grained control over network properties (notably packet loss) to stress-test the algorithm. The topology here is similar to the testbed (26 nodes, 4 hops). We use the accurate and extensive simulations in Cooja to reproducibly uncover trends and insights, which we then validate from the real deployment in Flocklab.

Data set: As our source of data values, we use the Intel Lab data set [1], commonly used in the WSN literature. It contains temperature, light, humidity and voltage readings from 54 sensors, placed inside an indoor lab, over a period of 36 days. Nodes take a new reading every 31 seconds.

We select 26 nodes (the ones with IDs 22-47) that have good quality of readings and use their temperature values. The temperature values follow a periodic cycle with the temperature rising during the day and falling moderately during the night, partly controlled by the heating and ventilation system. Out of the 36 days, we use the first day of readings for the experiments on Flocklab and in Cooja (20 hours). In our experiments, we replay the temperature values from the data set at each node, using the same sampling frequency.

Monitoring functions: We experiment with both linear and non-linear monitoring functions, namely the global *average* and *variance* of the sensor readings. The latter is of practical use in many different scenarios, e.g. to detect the presence of an area with irregularly high temperatures i.e. a hot-spot. The former is a simple case that can be solved even without the need for geometric monitoring but we include it for completeness. For the average we choose a threshold of $T = 20°C$ and for the variance a threshold of $T = 2°C^2$. which is crossed twice during the daily cycle. Similar to Sharfman et al. [25], we track the variance of readings in the GM framework, by having nodes tracking a local

statistics vector $\vec{v} = (t, t^2)$ where t is the current temperature reading. Unless stated otherwise in the description of the experiment, the variance will be used.

Experiments: Table 6.1 summarizes the different experiments presented in this section. As a comparison, we adopt the *baseline* method (also used in [24]) where nodes broadcast at every *epoch*, i.e. every sensor reading, as soon as they get it. We choose this baseline since: (i) there is no other method but GM that solves the problem of threshold monitoring for the general case (apart from its variants, discussed in Section 6.6) and (ii) it allows us to directly compare the theoretical savings with the ones achieved in practice (c.f. Section 6.5.1).

Metrics: For each of the experiments mentioned above, we are interested in the following metrics:
(i) *Communication reduction* achieved by GM, measuring the number of updates propagated; it is a measure of the efficiency of GM, purely from the application point of view.
(ii) *Duty cycle (DC)* i.e. the fraction of total time that a node has its radio turned on, computed using Contiki's power profiler [6]. We also define the *lifetime improvement* achieved through GM, as the reduction in duty cycle achieved through the execution of GM, compared to the baseline.
(iii) *Loss rate* measured as the number of individual destinations of a packet that fail to receive it (e.g. if a node sends an update and only 24 out of the other 25 nodes receive it, the loss rate is 1/25).
(iv) *Communication Latency* is the average time that a packet originating from node A needs to reach another node B.
(v) *Accuracy* and *Responsiveness* where the later relates to the latency of detecting an actual threshold violation. See respective definitions later in Section 6.5.4.

6.5 Evaluation from a holistic system perspective

6.5.1 Full-system simulations

In the full system simulation (Table 6.1, A), 20 hours of data are used for each configuration. We collect results both for the geometric method and for the baseline. In every configuration, the GM method achieves **4.31 times** communication reduction when monitoring the variance (Table 6.2, col 9) i.e. only 23.2% of the sensor readings are actually propagated.

We start by discussing the impact of the *channel check rate (CCR)*. Table 6.2 summarizes the results for a channel check rate ranging from 8 to 64 Hz, averaged across all 26 nodes.

Intel Lab Dataset (20 hours), Monitored Functions: **Variance** (Threshold T=2) and **Average** (Threshold T=20)													
Channel Check Rate (CCR) in Hz	Baseline (variance/average)			GM, Function: variance					GM, Function: average				
	Duty cycle	Loss rate (%)	Latency (ms)	**Duty cycle**	Loss rate (%)	Latency (ms)	Lifet. Impr.	Comm. Red.	**Duty cycle**	Loss rate (%)	Latency (ms)	Lifet. Impr.	Comm. Red.
8	8,44	9,98	18348	2,68	1,19	3862	3,15x		0,76	0,46	3716	11,04x	
12	7,73	0,59	760	2,57	0,22	590	3,01x		1,03	0,41	2465	7,50x	
16	7,28	0,18	314	2,70	0,11	336	2,69x		1,37	0,02	2078	5,31x	
24	6,88	0,07	189	3,11	0,07	212	2,21x	4,31x	1,96	0,20	1078	3,51x	44x
32	7,26	0,04	162	3,70	0,04	190	1,96x		2,68	0,97	906	2,71x	
48	7,81	0,12	139	4,88	0,12	151	1,60x		3,85	0,45	581	2,03x	
64	9,11	0,14	134	6,18	0,12	157	1,47x		5,31	1,01	703	1,71x	

Table 6.2: **Full system simulations**: Duty cycle, loss rate and latency for the GM method vs the baseline, with varying CCR. The baseline method behaves the same way, regardless of the function.

Flocklab Testbed, Monitored Function: **Variance**									
		Duty Cycle (%)		Comm. Reduction	Lifet. Improv.	Loss rate (%)		Latency(ms)	
CCR	Dataset section	GM	Baseline			GM	Baseline	GM	Baseline
12	low comm. (0:00 - 03:00)	1,33	6,98	45,74x	5,26x	0,45	5,43	908	1704
	high comm. (8:00 - 12:00)	4,28		2x	1,63x	0,88		634	
24	low comm. (0:00 - 03:00)	2,05	6,42	45,74x	3,13x	0,91	0,84	223	268
	high comm. (8:00 - 12:00)	5,50		2x	1,17x	0,49		262	

Table 6.3: **Testbed experiments**: Results from the Flocklab testbed, on two 3 hour periods from the dataset.

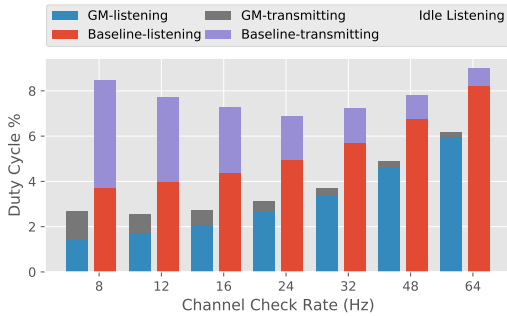


Figure 6.2: **Full system simulations**: The duty cycle, broken down to sending and listening, as well as the cost of idle listening.

Loss rate & Latency: It is useful to mention here that as GM reduces the amount of data sent, it also reduces the possibility of losses. GM has slightly higher latency at CCR values higher than 16 Hz; this is due to the extra, small amount of processing that is added at every hop. Reducing the processing cost (using our approximation) from 20ms to 2ms is important here for two reasons:(i) the overall latency decreases (ii) there is enough time for processing between packet receptions (for CCR value of 64 Hz, nodes wake up to receive every 16ms).

Duty cycle: We next turn our attention to duty cycle, the metric that is directly related to a node's effective lifetime. GM results in significant reduction in duty cycle. As an example (Table 6.2, col 5, CCR=12 Hz), using GM reduces the duty cycle from 7.73% to just 2.57%, a three-fold improvement. However, this improvement diminishes as the CCR increases. The lifetime improvement between the best configurations is 2.8x (compare duty cycles between 12 Hz for GM and 24 Hz for the baseline), which is far from the 4.31x communication reduction achieved by the method.

A brief look at the respective results from monitoring the *average* (Table 6.2, col 10), shows that the effects mentioned above for the variance are even more pronounced now. In this case, GM manages to reduce communication by 44 times, keeping nodes mostly quiet throughout the execution. In terms of duty cycle, GM reduces it by an impressive amount (up to 11 times for a CCR value of 8 Hz), but still, 4 times less than the achieved reduction in communication.

Duty cycle decomposition: A detailed look on the duty cycle *explains the aforementioned differences*. Figure 6.2 shows the duty cycle for GM and the baseline method (when tracking the variance), as well as its individual components. First, the percentage of the duty cycle that is spent on sending is

greatly reduced using the GM method, directly matching the communication reduction ratio achieved by the algorithm. Subsequently, the GM version spends less time receiving data at each node. Also, notice that the time spent on transmitting decreases as the channel check rate grows. This is simply because broadcasts are shorter when the CCR is high. In this figure, we have also included the cost of idle listening, i.e. the cost of turning on the ratio periodically to check for traffic, even though there is nothing to receive. This cost is the same for both methods and is computed from the CCR value. It is evident that this cost dominates the duty cycle when the channel check rate increases. Even for small CCRs, *the idle cost represents a significant overhead*, that reduces the potential lifetime savings of the algorithm.

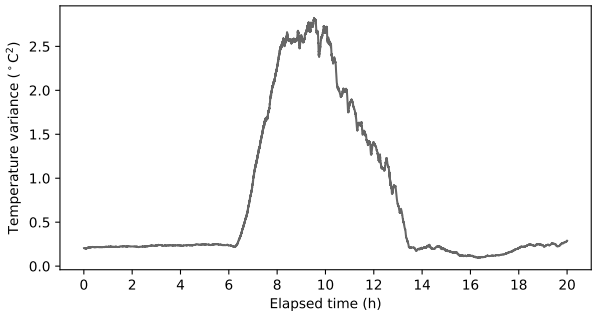
6.5.2 Validation through testbed experiments

We use the testbed experiments of this section as a way to validate the insights and trends gained from the full-system simulations that we presented above.

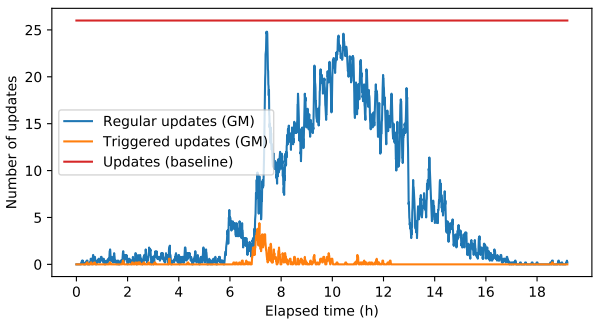
Experiment settings: In this section, we present the results from the execution on the Flocklab testbed (Table 6.1, B). Due to usage restrictions on the testbed, we do not replay full day measurements from the dataset. Instead we focus on sections of particular interest. We select two sections from the data set, 3 hours each (midnight and morning) where, as we detail further in the next experiment series, the communication pattern of the GM method is expected to be very different. For these experiments, we have picked a channel rate of 12 Hz, where the GM method had the lowest duty cycle on the simulations, as well as a rate of 24 Hz for comparison.

Results: Table 6.3 shows the overall results for the two sections of the dataset.[4] The topology of the testbed is slightly different than the one used in the simulation (more sparse), so the absolute values are different than Table 6.2, but we expect the general trends to hold. On the morning section (08:00 to 12:00), GM communicates 2 times less than the baseline. The lifetime improvement follows closely, and the duty cycle is reduced by 1.63 times. For the midnight section (00:00 to 03:00), GM achieves remarkable communication savings, reducing the number of readings that need to be propagated by 46 times, but the associated lifetime improvement is more modest (5.26 times). This indicates that, during this section (and unlike the previous one), even a check rate of 12 Hz is excessively high, and *most of the energy is spent on idle listening*. Similar results

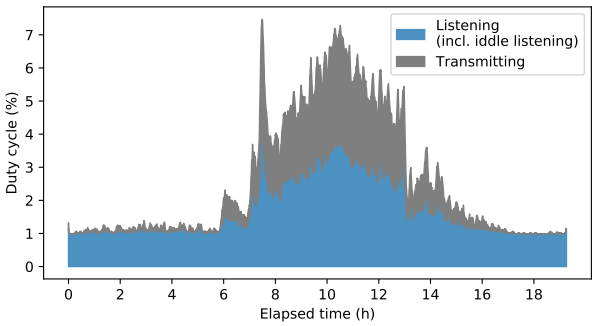
[4]The baseline version has the same communication behaviour (every reading gets propagated) regardless of the data section, so we evaluate it only on the midnight section.



(a) Variance of the temperature between nodes.

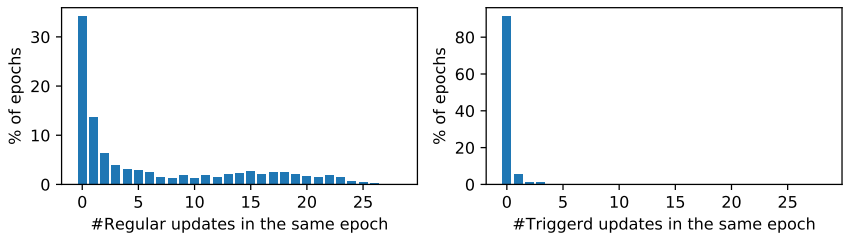


(b) Number of updates per epoch.



(c) The average duty cycle across nodes, during the execution.

Figure 6.3: **Runtime insights** from the execution of GM over a period of 20 hours.



(a) Regular updates.

(b) Triggered updates only.

Figure 6.4: **Runtime insights:** Percentage of epochs with concurrent updates (regular on left figure and triggered on the right)

can be seen when CCR is set to 24 Hz. Here, the lifetime improvements decrease for both data set sections, especially for the period with low communication (3,13x lifetime improvement).

6.5.3 Runtime insights: a closer look

We now take a closer look into a single experiment and provide insights for the communication behaviour of the algorithm (Table 6.1, C). We set the CCR to 12 Hz (that resulted in the best duty cycle for the GM case) and elaborate on detailed observations from GM, in order to distill deeper insights about the interplay between GM and the communication stack.

Monitored value: Figure 6.3a shows the actual value that is being monitored: the variance of the temperature readings. Due to variation in the temperature between different rooms during working hours, the data set exhibits a period of approximately 7 hours where readings between nodes have increased variance, up to 2.8 $°C^2$.

Update decomposition: Next, we count the number of updates that nodes propagate at every *epoch* of the execution. Recall that the epoch is defined as the sampling at which nodes take new measurements (for the Intel Lab data set this period is 31 seconds) and that the baseline method broadcasts all of them. For the GM method, we distinguish between two kinds of updates. *Regular updates* happen when a node gets a new sensor reading, detects a threshold violation and therefore decides to propagate this reading to all the other nodes. *Triggered updates* happen when a threshold violation is not caused by a new reading, but by an update received from a neighbour. Triggered updates are interesting from the communication protocol point of view, because they cause traffic bursts where

nodes want to concurrently share information across the network.

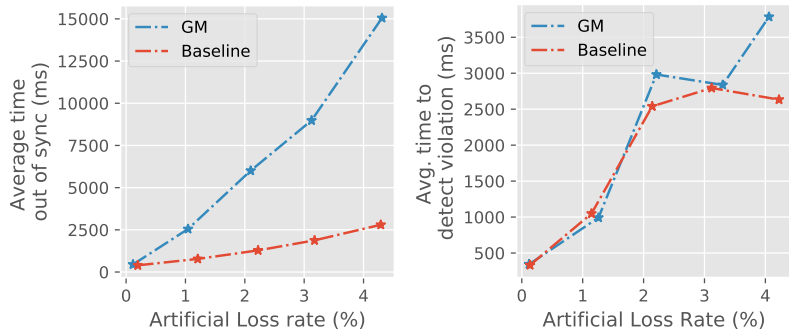
Temporal variation in comm. reduction: In Figure 6.3b we show the number of updates per epoch, for the GM method as well as the baseline, computed over a 1.5 minute sliding window and averaged across all nodes. Note that the x-axis has been translated to reflect the elapsed time in hours, in order to match Figure 6.3a. The baseline induces the same number of updates per epoch, equal to the number of nodes. On the contrary, the GM method significantly reduces the number of sensor readings that need to be updated per epoch. Especially during the periods when variance is small and away from the threshold, almost all communication is suppressed. As the variance comes closer to the specified threshold, nodes start detecting frequent violations and update more of their readings. We can also see a period close to the threshold where these updates trigger violations on other nodes, as well as a peak in the number of updates when the actual value of the variance is close to the threshold of 2 $^{\circ}C^2$.

Figure 6.4 presents a different view of the same results. Here, we count the percentage of epochs with *concurrent* updates, both regular and triggered. Figure 6.4a shows that a significant amount of epochs (34%) do not contain any regular updates, but the majority of epochs contain concurrent updates. Also, *a noticeable percentage of epochs contain many concurrent updates* (15-26 concurrent updates). Triggered updates occur very rarely in this data set (Figure 6.4b): 91% of epochs do not have any triggered updates. However, there are epochs where several updates are triggered concurrently.

Temporal variation in duty cycle: In Figure 6.3c we take another look at the duty cycle and monitor how it changes during the execution. Here, the duty cycle is computed at every epoch and averaged across all nodes. The duty cycle follows the same trend as the number of updates in Figure 6.3b: at periods where communication is high, the radio needs to stay on longer in order to send or receive the extra traffic. From Figure 6.3c, it is also evident that *the idle listening cost is a dominant factor* that affects the duty cycle and limits the potential of the GM method: even during periods with no activity, nodes waste a constant amount of time to check the radio for transitions.

6.5.4 Accuracy/Responsiveness: the effect of packet losses

In the following, we will state that a node A is "out of sync" with respect to node B, when its global estimate is different from node B, either: (i) because an update from node B is still "in flight" or (ii) because an update from node B was lost. In the first case, node A will stay out of sync until the "in flight" update arrives. In the second case node A will stay out of sync until a later update from



(a) Average time (in ms) that nodes are out of sync

(b) Average time (in ms) to detect that the threshold was crossed.

Figure 6.5: **Accuracy/Responsiveness** of the algorithm, measured as we introduce packet losses.

B successfully arrives.

Define as *accuracy* the average time a node is out of sync with respect to any other node. Define also the *responsiveness* of the algorithm as the average time elapsed from the moment the value of the monitored function has crossed the predefined threshold, until the moment a node actually detects this. Note that accuracy is just a measure of the time nodes have stale information with respect to each other, which might not necessarily be an issue if the network aggregate is not close to the threshold. On the contrary, responsiveness captures the *critical time* during which the threshold has been actually crossed and the node has not yet detected it.

We next evaluate both metrics (cf. Table 6.1, D). We run simulations where we intentionally introduce packet losses, with a controlled rate, at each node and report the resulting loss rate. The CCR value is set to 16 Hz.

Figure 6.5a reports the average duration that a node is out of sync, for increasing loss rates. For small loss rates (0.1% approximately) nodes are out of sync mostly until "in flight" packets arrive (400 ms). Overall, both methods stay out of sync longer as we introduce packet losses, since more and more nodes will miss updates and will have to wait for at least one full epoch to get back in sync. GM is affected to a much greater degree, simply because some nodes will suppress their transmissions, keeping others out of sync for longer.

Figure 6.5b shows how the *responsiveness* of the two methods changes as we introduce losses. On low loss rates (0.1%) nodes in both version quickly

detect the threshold violations, within approximately 300 ms. The time to detect the violation increases rapidly for both versions as more losses are introduced. Even with a little over 1% loss rate, it takes 1 second on average to detect the violation, for both versions. On higher loss rate values, GM seems to take longer to detect the violation, since more and more nodes are out of sync and have stale information. Note that the gap between GM and the baseline is not as large for responsiveness as for accuracy. This is because, close to when the threshold violation happens, nodes in GM detect local violations frequently, so the behaviour becomes similar to the baseline, for a short time interval, until the threshold is exceeded.

Overall summary of results: The results show how GM is of practical use in IoT environments. The presented extensive simulations and testbed runs' outcomes suggest that GM can indeed bring several fold reduction to duty cycle. However, the full system perspective reveals that the benefits achieved in practice can be far from the expected ones, due to the overhead of idle listening. Also shown in the evaluation is the insight that to avoid that cost is hard in GM, due to changes in the communication pattern: during its execution, GM has periods of little to no activity that would benefit from a low CCR, as well as periods with high traffic that need a higher CCR value. This opens interesting questions as to whether approaches that dynamically choose CCR values [20, 21] would be efficient in managing the data-dependant communication pattern of GM. Finally, we show that packet loss has a direct impact on the accuracy and responsiveness of the algorithm, suggesting that the network stack should be tuned to ensure a low loss rate at all times, especially during the critical periods where the global estimate crosses the threshold. This also suggests that, for applications with no strict requirements on responsiveness, it might be beneficial to sacrifice some responsiveness in favor of a lower duty cycle (e.g. by relaxing the "eagerness" of the propagation protocol).

6.6 Other related work

Geometric Monitoring (GM): In [10], the authors orthogonally augment GM with sketches, that further reduce the communication cost by keeping track of an approximation on the network-wide aggregate. Giatrakos et al. [11] combine the communication reduction of GM with prediction models that track the temporal evolution of sensor readings and only report when the model needs to be updated. A summary of the use of GM for query tracking in distributed streaming systems can also be found in [9]. This interest has been motivating also for the work in our paper.

In [15], the authors introduce shape sensitive geometric monitoring, that takes into account properties of the monitored function. Lazerson et al. [17] propose a variant of GM that addresses the computational complexity of checking for violations. They introduce a method of approximating the monitored function to convex/concave components, so that it can be checked for violations fast, without the need to construct the respective spheres. The approximation we propose in our work is orthogonal, in the sense that we leave the function intact and instead bind the local area that nodes have to keep track of for violations.

GM has been studied in the general context of WSN from a high level perspective [2, 25]. In [25] the authors present an adaption of GM that is designed for clustered topologies. In [2] GM is used for detecting outliers in the readings of wireless sensor nodes. In both of these lines of work, the network is only considered as a communication abstraction and practical systems aspects are not considered. Differently, we take a full system perspective and consider practical aspects such as the effect of the RDC protocol and packet loss.

Data prediction and aggregation for WSNs: For data aggregation in WSNs, usually the goal is to collect at a single, sink node, the sensor readings from every other node in the network. A significant amount of research effort has been put on reducing communication by aggregating data along the way from sources to a destination [13, 16, 19].

Data prediction extends this design space by building models that approximate (within some error guarantees) the sensed values at every node. An update is only sent to the sink when a node's value deviates from the ones predicted by the model [3]. For example, Raza et al. [23] propose a data prediction method and are the first to evaluate it on real sensor nodes. Similar to the spirit of our work and through a full system evaluation, they find that the sleeping interval of the MAC protocol and the cost of maintaining a topology have significant effects on the practical lifetime of the nodes. Istomin et al. [14] extend these findings and propose Crystal, a protocol based on synchronous transmissions that is tailored for the communication requirements of data prediction.

Differently from data prediction, in GM there is no need to model the physical quantities sensed at the nodes, which is often challenging and requires expert knowledge. Moreover, in data prediction, nodes model their stream of readings locally without dependencies on neighbouring nodes, whereas the threshold monitoring problem that GM addresses is inherently distributed, hence the work in [23] and [14] is not readily applicable to distributed threshold monitoring.

6.7 Conclusions and future work

Inspired by important results on the problem of continuous monitoring, we take a full-system approach on the applicability of Geometric Monitoring (GM) on real IoT environments. In particular, we focus on processing and communication, as well as a cross-layer perspectives. We provide a method that simplifies the threshold checking for resource-constrained IoT devices, with an approximation that is particularly useful for many common cases. We also study the GM-communication interplay: we confirm several-fold reduction in communication which in turn leads to battery lifetime improvements on the nodes. We observe however that the resulting improvement falls short of the theoretical savings, which, as our results underline, is due to the baseline energy overhead of the network stack. Moreover, we show that packet losses have a magnified effect on the accuracy and responsiveness of the algorithm. Both the above motivate cross-layer approaches for practical purposes. We expect that these insights will enable the design of custom protocols and cross-layer optimization techniques that will unlock the full potential of GM threshold monitoring applications for IoT systems.

Bibliography

[1] Peter Bodik, Wei Hong, Carlos Guestrin, Sam Madden, Mark Paskin, and Romain Thibaux. Intel Lab Data, 2004.

[2] Sabbas Burdakis and Antonios Deligiannakis. Detecting Outliers in Sensor Networks Using the Geometric Approach. In *IEEE International Conference on Data Engineering*, 2012.

[3] Amol Deshpande, Carlos Guestrin, Samuel R. Madden, Joseph M. Hellerstein, and Wei Hong. Model-driven data acquisition in sensor networks. In *30th Int'l Conf. on Very Large Data Bases*, VLDB '04, 2004.

[4] Adam Dunkels. The ContikiMac radio duty cycling protocol. Technical report, Swedish Institute of Computer Science, 2011.

[5] Adam Dunkels, Björn Gronvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *29th IEEE Int'l Conf. on Local Computer Networks*, 2004.

[6] Adam Dunkels, Fredrik Osterlind, Nicolas Tsiftes, and Zhitao He. Software-based on-line energy estimation for sensor nodes. In *4th Workshop on Embedded Networked Sensors*, EmNets. ACM, 2007.

[7] Laura Feinstein, Dan Schnackenberg, Ravindra Balupari, and Darrell Kindred. Statistical approaches to DDoS attack detection and response. In *Proc. DARPA Information Survivability Conference and Exposition*, 2003.

[8] Zhang Fu, Magnus Almgren, Olaf Landsiedel, and Marina Papatriantafilou. Online temporal-spatial analysis for detection of critical events in cyber-physical systems. In *IEEE Int'l Conf. on Big Data*, pages 129–134. IEEE, 2014.

[9] Minos Garofalakis. Approximate geometric query tracking over distributed streams. *IEEE Data Eng. Bull.*, 2015.

[10] Minos Garofalakis, Daniel Keren, and Vasilis Samoladas. Sketch-based Geometric Monitoring of Distributed Stream Queries. *Proc. VLDB Endow.*, 2013.

[11] Nikos Giatrakos, Antonios Deligiannakis, Minos Garofalakis, Izchak Sharfman, and Assaf Schuster. Prediction-based geometric monitoring over distributed data streams. In *ACM SIGMOD International Conference on Management of Data*, 2012.

[12] Vincenzo Gulisano, Magnus Almgren, and Marina Papatriantafilou. Metis: a two-tier intrusion detection system for ami. In *Int'l Conf. on Security and Privacy in Comm. Sys.*, pages 51–68. Springer, 2014.

[13] Vincenzo Gulisano, Magnus Almgren, and Marina Papatriantafilou. When smart cities meet big data. *Smart Cities*, page 40, 2014.

[14] Timofei Istomin, Amy L. Murphy, Gian Pietro Picco, and Usman Raza. Data Prediction + Synchronous Transmissions = Ultra-low Power Wireless Sensor Networks. In *Proc. of the ACM Conf. on Embedded Network Sensor Sys.*, SenSys, 2016.

[15] Daniel Keren, Izchak Sharfman, Assaf Schuster, and Avishay Livne. Shape Sensitive Geometric Monitoring. *IEEE Trans. Knowledge and Data Eng.*, 2012.

[16] Bhaskar Krishnamachari, Deborah Estrin, and Stephen Wicker. The impact of data aggregation in wireless sensor networks. In *W'shops 22nd Int'l Conf. on Distr. Computing Sys.*, 2002.

[17] Arnon Lazerson, Daniel Keren, and Assaf Schuster. Lightweight monitoring of distributed streams. In *ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*, 2016.

[18] Roman Lim, Federico Ferrari, Marco Zimmerling, Christoph Walser, Philipp Sommer, and Jan Beutel. Flocklab: A testbed for distributed, synchronized tracing and profiling of wireless embedded systems. In *2013 ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2013.

[19] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, December 2002.

[20] Christophe J. Merlin and Wendi B. Heinzelman. Duty cycle control for low-power-listening mac protocols. In *5th IEEE Int'l Conf. on Mobile Ad Hoc and Sensor Systems*, 2008.

[21] Xu Ning and Christos G. Cassandras. Optimal dynamic sleep time control in wireless sensor networks. In *2008 47th IEEE Conf. on Decision and Control*, 2008.

[22] Fredrik Osterlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, and Thiemo Voigt. Cross-level sensor network simulation with cooja. In *31st IEEE Conf. on Local Computer Networks*, 2006.

[23] Usman Raza, Alessandro Camerra, Amy L. Murphy, Themis Palpanas, and Gian Pitero Picco. Practical Data Prediction for Real-World Wireless Sensor Networks. *IEEE Trans. on Knowledge and Data Eng.*, 2015.

[24] Izchak Sharfman, Assaf Schuster, and Daniel Keren. A geometric approach to monitoring threshold functions over distributed data streams. In *ACM SIGMOD Int'l Conf. on Management of Data*, 2006.

[25] Izchak Sharfman, Assaf Schuster, and Daniel Keren. Aggregate threshold queries in sensor networks. In *IEEE Int'l Parallel and Distributed Processing Symp.*, 2007.

PAPER VI

Charalampos Stylianopoulos, Magnus Almgren,
Olaf Landsiedel, Marina Papatriantafilou

Continuous Monitoring meets Synchronous Transmissions and In-Network Aggregation

An adapted version of the paper that appeared in *the Proceedings of the 15th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pp. 157-166, IEEE 2019.

7

Continuous Monitoring meets Synchronous Transmissions and In-Network Aggregation

Abstract

Continuously monitoring sensor readings is an important building block for many IoT applications. The literature offers resourceful methods that minimize the amount of communication required for continuous monitoring, where Geometric Monitoring (GM) is one of the most generally applicable ones. However, GM has unique communication requirements that require specialized network protocols to unlock the full potential of the algorithm.

In this work, we show how application and protocol co-design can improve the real-life performance of GM, making it an application of practical value for real IoT deployments. We orchestrate the communication of GM to utilize the properties of a state-of-the-art wireless protocol (Crystal) that relies on synchronous transmissions and is designed for aperiodic traffic, as needed by GM. We bridge the existing gap between the capabilities of the protocol and the requirements of GM, especially in the case of periods of heavy communication. We do so by introducing an in-network aggregation technique relying on latent opportunities for aggregation that we exploit in Crystal's design, allowing us to reliably monitor a wide spectrum of aggregate functions, including duplicate-sensitive ones, such as sum, average or variance. Our results from testbed experiments with a publicly available dataset show that the combination of GM and Crystal results in a very small duty-cycle, a 2.2x - 3.2x improvement compared to the baseline and up to 10x compared to previous work that uses a mainstream network stack. We also show that our in-network aggregation technique reduces the duty-cycle by up to 1.38x.

7.1 Introduction

The problem of monitoring a network-wide system state has fundamental uses in Wireless Sensor Networks (WSN). Whether we are monitoring the temperature in a room to detect outliers [3] and hot-spots [28] or machinery operation statistics in a factory to detect malfunctions [21], we are often interested in keeping track of a function calculated over all the network's readings. More specifically, given a set of sensor nodes $n_1, ..., n_N$ with readings $\vec{v_1}, ..., \vec{v_N}$ that vary over time, we want to continuously track whether the value of a function f, defined over the network-wide weighted average of the readings, is higher (or lower) than a predefined threshold T.

One of the most generally applicable solutions that addresses the communication complexity of the problem is Geometric Monitoring (GM) by Sharfman et al. [27]. GM is a method that can monitor any function, linear or not, with

respect to a threshold and can suppress most of the readings without having to communicate. Since its introduction, GM has been extended with sketches [10] and prediction models [11], showing that it is extensible and applicable in many scenarios.

When considering the applicability of GM for IoT deployments, there are aspects to consider that go beyond the algorithm itself. The communication pattern of GM is highly data-dependent and varies significantly during run-time. This implies a challenge for most network protocols as they are usually optimized for periodic communication. Moreover, the actual energy savings on the nodes depend on the underlying communication stack and the way it interacts with GM. Previous work [29] studied the performance of GM in an applied IoT context and showed that the energy overhead of mainstream network stacks limits the effectiveness of the algorithm in practice. Thus, it is interesting to study how to close the gap between the requirements of GM and appropriate protocols that can support them.

Crystal [13, 14] is a recently introduced wireless protocol based on synchronous transmissions that is specifically designed to favor aperiodic communication patterns: it efficiently handles communication when there is little to send within an epoch, but can also accommodate epochs with heavy communication. However, Crystal was originally designed with applications featuring aperiodic data collection from multiple senders to a single sink node. Contrary, GM requires that any node should be able to broadcast updates to every other node in the network. Moreover, even though Crystal gracefully handles the case of low communication load, it misses opportunities to save energy in cases of high load that can benefit, e.g., from in-network aggregation.

In this work, we bridge this gap between the requirements of GM as an application and Crystal as a state-of-the-art wireless protocol that relies on synchronous transmissions. By doing so, we provide a practical realization of a system that continuously monitors sensor values with a high degree of communication suppression (due to properties of GM) and operates at low duty-cycle with high reliability (due to properties of Crystal). We also propose *Arctium*, adding in-network aggregation to synchronous transmission protocols for aperiodic communication; this orthogonal design allows any application that monitors aggregated values from sensors to get additional reduction in communication on top of the existing design of Crystal. Hence, we show how to: (i) make algorithms such as GM practical for real IoT deployments, in combination with modern protocols such as Crystal and (ii) reduce the energy consumption of such algorithms even further by adding in-network aggregation.

In particular, we make the following contributions:

- We orchestrate and deploy Geometric Monitoring on top of a state-of-the-art wireless network protocol that relies on synchronous transmissions.
- We introduce *Arctium*,[1] a novel method for in-network aggregation that relies on latent opportunities for aggregation in Crystal's design and reliably tracks aggregates, including duplicate-sensitives ones, using fewer transmissions than Crystal.
- We evaluate the effects of application and protocol co-design on real IoT nodes, using publicly available data-sets and show a reduction in energy consumption of up to 10x compared to previous work that uses a mainstream network stack, and how in-network aggregation further reduces it by 1.13-1.38x.

The paper is organized as follows. Section 7.2 summarizes GM and Crystal. In Section 7.3, we present our GM and Crystal co-design and describe *Arctium*, our aggregation scheme. In Section 7.4, we show the results from our evaluation. We present related work in Section 7.5 and conclude in Section 7.6.

7.2 Preliminaries

7.2.1 The Geometric Monitoring Method (GM)

In their seminal work [27], Sharfman et al. present a general method able to monitor (with respect to a threshold) arbitrary functions defined over network-wide aggregates. Instead of having nodes communicate for every new reading, each node uses local information to decide whether to send an update. We now give a summary of the Geometric Monitoring method. A more detailed analysis, as well as the proofs can be found in the original publication by Sharfman et al. [27].

Assume a network of N nodes, with sensor readings $\vec{v_1}, ..., \vec{v_N}$, called *local statistics vectors*. Those vectors consist of one or more variables that each node monitors and vary over time. These vectors are only known locally, but sporadically a node n_i will broadcast its $\vec{v_i}$ to every other node. The last broadcasted value from n_i is denoted as $\vec{v_i'}$. The weighted average of the local vectors is called the *global statistics vector*.

[1] Arctium is a genus of plants, notable for their velcro-like heads that tend to stick and aggregate on other materials.

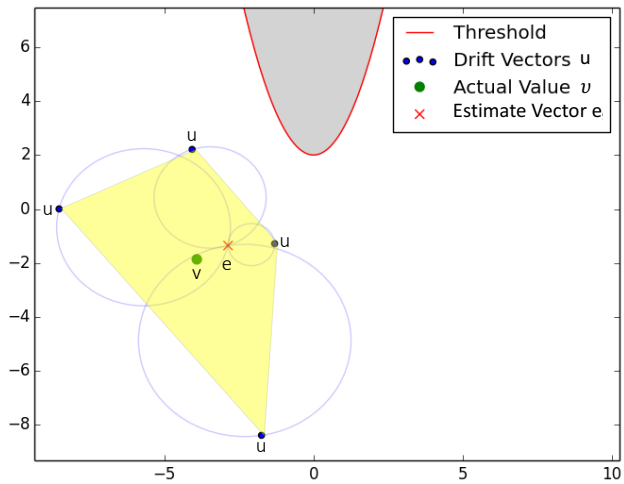


Figure 7.1: An example illustrating the GM method. The method keeps track of whether the actual value (green dot) is over or under the threshold (red line), by monitoring the circles formed by the drift vectors (gray circles).

$$\vec{v} = \sum_{i=1}^{N} w_i * \vec{v_i} \tag{7.1}$$

Similarly, the weighted average of the last broadcasted values is called the *estimate vector* ($\vec{e}$) and it is known to all nodes. Given a function f and a threshold T, we want to continuously monitor whether or not the value $f(\vec{v})$ is under the threshold. Equivalently, we want to always know whether or not $\vec{v}$ lies in an are where the function takes values below the threshold.

When a node measures a new set of sensor readings, its local statistics vector will drift ($\vec{\Delta v_i} = \vec{v_i} - \vec{v_i'}$). The *drift vector* $\vec{u_i}$, defined as the displacement of the estimate vector because of the new drift, i.e. $\vec{u_i} = \vec{e} + \vec{\Delta v_i}$, can be computed locally without communication. Figure 7.1 shows an example of the method, also depicting the values defined above.

The convex hull of the drift vectors (yellow area) is defined as the set of all the convex combinations of $\vec{u_i}$ ($\sum \theta_i \vec{u_i}$). As such, it is clear that the weighted average of the drift vectors (defined similarly to Equation 7.1) would be part of this set. With simple substitution, one can also see that the global statistics vector is equal to the weighted average of the drift vectors and thus it must also

lie in the convex hull of the drift vectors. Thus, as long as the convex hull does not cross the threshold (i.e. lies in the white area), $\vec{v}$ is also guaranteed to not have crossed the threshold. However, nodes cannot locally determine the convex hull, as that would require knowing all $\vec{u_1}, ..., \vec{u_N}$.

This is where the final part of the method comes into play. Let each node create a sphere locally, centered at $\frac{\vec{e}+\vec{u_i}}{2}$ with a radius of $\frac{\vec{e}-\vec{u_i}}{2}$. This is possible since $\vec{u_i}$ is known to n_i and $\vec{e}$ is the same across all nodes at a given time. Sharfman et al. [27] prove that the union of those spheres strictly covers the convex hull. Therefore, a node only needs to track whether its locally computed sphere crosses the threshold. If yes, it will send its local vector to everyone, subsequently updating the estimate vector; else, it can remain quiet.

What is relevant for this work, from a *communication behavior* point of view, is that the method results in nodes communicating aperiodically: there are intervals when most nodes suppress their readings with almost no communication, while at other times multiple nodes need to broadcast their values so that all nodes in the network can update their global estimate. Such a dynamic communication pattern makes GM a challenging application for many state-of-the-art low-power wireless protocols.

7.2.2 Crystal

Crystal [13, 14] is a protocol for low power and highly reliable aperiodic collection of data from multiple nodes to a single sink. Crystal has *epochs* and is designed to be particularly useful for applications where the number of nodes that have data to be collected at each epoch varies, but is typically low. In its core, Crystal relies on Glossy [7], the seminal work of Ferrari et al. By tight (sub-microsecond) scheduling of transmissions, Glossy makes use of the *capture effect* [22] and *constructive interference* [7] to achieve network-wide flooding with extremely small latency.

Crystal, in turn, builds a schedule on top of Glossy that consists of a series of phases that form an *epoch*. Figure 7.2 shows an example of a single Crystal epoch for a network with three nodes about to send their values to the sink. In this example, the epoch takes nine phases until completion. V_X indicates the value that node X is trying to send to the sink. First, during the S-phase, the sink floods a synchronization message to ensure all nodes have a common reference point. Then, during the T-phases (phases 2, 4, 6 and 8), any node that has data to be collected initiates its own Glossy transmission. With high probability, one of the transmissions reaches the sink, which acknowledges the transmission in the following A-phase. In this example, the value from Node C reached the sink at phase 4 and was acknowledged at phase 5. Nodes that did

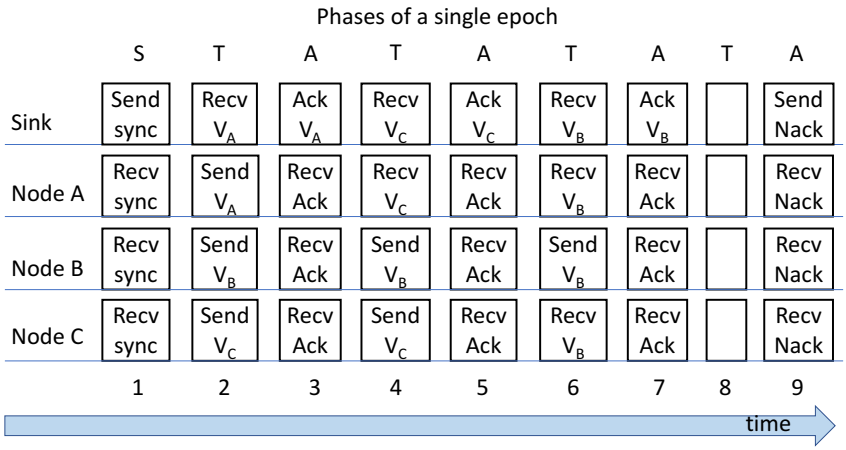


Figure 7.2: A direct adaptation from [14], showing the different phases of Crystal, for a network with three nodes and one sink. Each box indicates a (possibly multi-hop) Glossy flood.

not get their transmissions acknowledged transmit again during the next T-phase, until one (or more, for safety) T-phases are empty. In this case, the sink issues a negative acknowledgment (phase 9) and the network can go back to sleep until the beginning of the next epoch.

7.3 GM, Crystal and *Arctium* co-design

7.3.1 Overview

The first idea is to use GM on top of the existing communication schedule followed by Crystal (see Figure 7.2), with benefits from both worlds: (i) communication reduction from using GM as an application, and (ii) efficient handling of the aperiodic communication pattern using Crystal as the underlying protocol. This can be achieved by using the T-phase to send updates from nodes who detected threshold violations and the A-phases to trickle down the changes to the global estimate. All nodes perform the computation required by GM (threshold checking) between the T-A pairs to decide whether to transmit or just forward packets.

The second idea is to gracefully handle the (so far overlooked) cases of multiple concurrent transmitters. We propose *Arctium* to extend the design of Crystal in a novel way by introducing in-network aggregation inside Crystal's

schedule. We do so by extending the role of the T-phases: during these phases, nodes do not only forward messages towards the sink, but they also have the opportunity to overhear neighbouring transmissions and combine these with their own. As a result, *Arctium* requires fewer messages to complete the epoch and is more energy efficient than Crystal. *Arctium* is orthogonal to GM and works for any application that is monitoring aggregates, such as *min, max, sum, average* and *variance*.

In the rest of this section, we first describe how we design GM in coordination with Crystal. Then we introduce *Arctium* and outline how it enables aggregation on top of Crystal.

7.3.2 Orchestrating GM communication with synchronous transmissions

Crystal was originally designed for data collection: sensor values from any node must be collected at a single, fixed sink. The original requirements of GM are however different:[2] when a node detects a local threshold violation and transmits its update, all other nodes must receive it to recompute the global estimate. In order to bridge the gap between the two models and fit the requirements of GM on top of the existing Crystal schedule, we make use of the Glossy transmission during the A-phase. That transmission not only acknowledges the reception of a node's value by the sink, but it also lets every node in the network know the new, updated value of the global estimate.

Overall, in our system, nodes first collect new sensor readings and perform threshold checking (see Section 7.2.1) at the beginning of the epoch. If they cross the threshold, they send their update during the T-phase. If an update reaches the sink, the sink updates the global estimate. The new global estimate will then trickle-down to the other nodes during the A-phase, piggybacked on the acknowledgment of the previous T-phase transmission. When nodes receive the new global estimate, they have until the next T-phase to perform the threshold checking again (based on the new global estimate) and decide whether they need to transmit their update. This continues until there are no further transmissions (2-3 empty T-phases) and the epoch ends. Note that, on rare cases, a packet loss during the A-phase might go undetected resulting in some nodes momentarily having an outdated global estimate. This is corrected in the next A-phase when

[2]There are models of GM that consider a single, coordinating node that is able to query values from specific nodes (using e.g. a unicast) and resolve threshold violations. However, since unicast is not part of Crystal's schedule, we do not consider such models in this work.

the sink disseminates the new global estimate.

Efficient threshold checking: As mentioned in Section 7.2.1, nodes need to perform the threshold checking required by GM between the T-A pairs in the protocol, which can be computationally challenging for IoT devices with very limited computing and energy resources as the threshold surface might have an arbitrary shape.

We address this by approximating the GM-spheres with a simpler shape that makes the computations significantly faster while ensuring that we do not introduce false negatives. As an example in 2D, the original spheres (now circles, similar to the example Figure 7.2 in Section 7.2.1) can be replaced with squares containing the former, which results in simpler boundary conditions (as it is simpler to check whether the sides of a square, rather than points on a circle, cross a surface). Naturally, there will be cases where the square check will report a violation even though the circle inside it does not actually cross the threshold, hence sacrificing communication reduction for ease of computation. This relaxation might not cope with high dimensions, but in many cases of monitoring statistics such as variance or correlations between nodes, it provides a simple and efficient solution which we have also experimentally verified (e.g. when monitoring the variance, the computation time decreases from 20ms to just 7ms on the TelosB platform, see Section 7.4). In Section 7.5 we discuss other alternatives in existing literature [20].

7.3.3 *Arctium*: enhancing Crystal with in-network aggregation

Motivation: Crystal is designed to efficiently handle the case where only a few nodes have data to transmit within the epoch. However, for many applications including GM, there are epochs where many or all the nodes in the network have data to transmit. For example, for GM this happens when the monitored value is close to the threshold. Hence, there is potential for improvement on a protocol level, which we exploit by proposing *Arctium*.

The following two observations have guided our work of *Arctium*. (i) In applications such as GM, the individual values by themselves are not interesting but rather only the aggregate (e.g. GM needs to keep track of the average of the values to compute the global estimate). (ii) During each T-phase of Crystal, only one value from one node reaches the sink, while the other concurrent senders fail.

The goal is to allow nodes that did not manage to reach the sink to aggregate their own values with the values of neighbouring nodes. Subsequently, these nodes try to send their aggregated values. If they reach the sink, the sink

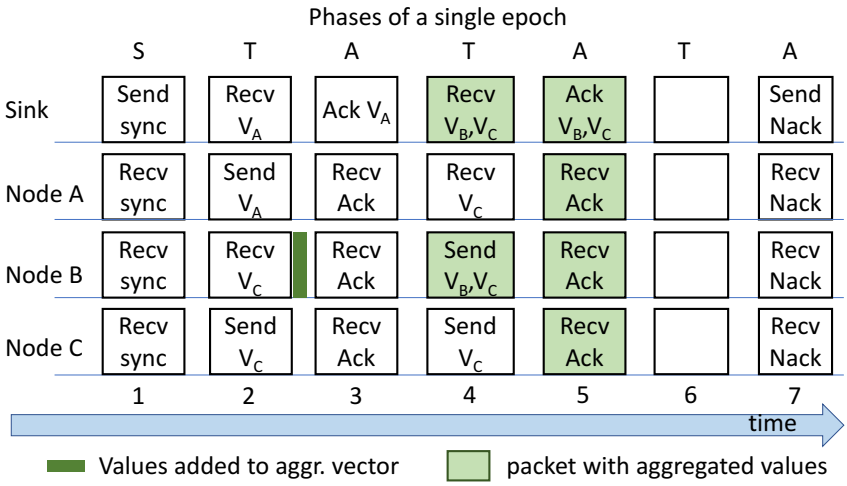


Figure 7.3: Example of an epoch with three nodes with values to send and one sink, using *Arctium*. During phase 2, node B overhears a transmission from node C and combines C's value with is own. Compared to Crystal in Figure 7.2, *Arctium* requires one less T-A pair to complete the epoch.

acknowledges multiple values in a single phase and the total number of phases in the epoch is reduced.

Fitting aggregation into Crystal: Figure 7.3 shows an example epoch using *Arctium*. Our aggregation scheme leverages the T and A communication phases of Crystal. If, during a T-phase, a node does not try to transmit a value of its own (Node B, phase 2), it can overhear a neighbouring transmission and propagate it. At the end of the T-phase, the value that was overheard is stored in a vector. The node then aggregates that value with its own or other, previously aggregated values and tries to send the aggregate to the sink. The sink in our system acknowledges the aggregated information during a single A-phase, in response to receiving an aggregated message (phase 5). As a result, the total number of phases in an epoch is reduced. In this case, two phases less than the Crystal example in Figure 7.2.

Keeping track of values from multiple nodes: To track which nodes' values we have aggregated, packets in *Arctium* hold, apart from the value, a small vector with the IDs of the nodes whose values are taken into account in the aggregate. The size of this vector determines the maximum number of nodes whose values we can aggregate in a single message. Even though this vector increases the size of the packet and adds overhead, we show in Section 7.4 that

even with a small vector (two or three IDs) we have good opportunities for aggregation without adding excessive overhead. A node also keeps a local vector with the IDs and values that are part of the aggregate. Keeping track of that vector allows nodes to add or remove values from the aggregate, based on the design choices described next.

Increasing the chances to aggregate: In order to give the opportunity for nodes that have values to transmit to overhear neighbouring transmissions and to combine them with their own, we introduce a *random back-off* during the T-phases. If a node has values to transmit, it will do so with a given probability p. Otherwise, the node will still be part of the T-phase but will only overhear and forward packets. Note that introducing such a back-off is not wasting opportunities for communication: only one node's value will reach the sink anyhow, so unless all nodes with data to send back-off simultaneously, there will be a successful reception at the sink.

Aggregation design choices: We now outline and motivate important design choices for the logic of when and how to aggregate. In the following, we denote with V_X the value that node X is trying to send to the sink.

Point 1: If node A overhears during a T-phase that its value V_A has been aggregated by node B, node A stops transmitting V_A and allows node B to send it instead. This helps to reduce the number of concurrent transmitters within each T-phase and reduces the size of the epoch.

Point 2: If node A, which has aggregated (V_A, V_B, V_C), hears an acknowledgment from the sink that contains (either standalone or part of an aggregated packet) the ID of B, A removes V_B from its vector and continues with the aggregated values of A and C. This allows us to keep the rest of the values that have been aggregated so far and only remove the ones that were acknowledged by the sink.

Point 3: Nodes do not combine one set of aggregated values with another. E.g., if node A with aggregated values (V_A, V_B, V_C), overhears a transmission from node D which has aggregated (V_D, V_B, V_E), node A will not attempt to combine the aggregates. This serves two purposes: (i) it avoids duplicate values (in the example above V_B would be counted twice) and (ii) it makes enforcing Point 2 above possible, because a node knows the individual contributions of the values that it is aggregating.

Point 4: If an aggregating node A overhears that node B is aggregating some similar IDs and B is a more successful aggregator (has a larger vector of aggregated values), node A will remove those common IDs from its own vector and allow B to take care of them. This helps to decrease the

probability of cases where two nodes each have aggregated values which they cannot combine, due to Point 3.

Algorithm 7.1 summarizes the pseudo-code of the aggregation logic, in correlation with the points described above.

Properties of *Arctium*: We now argue about the following three claims regarding the behavior of *Arctium*. For readability, we first present them considering absence of packet losses and then we discuss the implications incurred by packet losses.

Claim 7.1. *If Crystal delivers a value V_X to the sink, Arctium delivers it as well in the same epoch. In other words, Arctium guarantees delivery of all values.*

Arctium makes sure no values are dropped in the following way. Consider a node A trying to send a value V_A to the sink. First, node A will not stop trying to send V_A until it is acknowledged by the sink or picked up for aggregation by another node B (see point 1). Second, if a node B has aggregated the value V_A of node A, node B will not remove it from its aggregation vector until it is acknowledged by the sink (see point 2) or it is picked up by a more successful aggregator (see point 4). In any case, until the value V_A is acknowledged by the sink, there is at least one node (either the originator of that value or the node(s) that aggregated it) that is responsible for that value and keeps trying to send it to the sink.

Claim 7.2. *Arctium guarantees no duplicates to the sink.*

This is because: (i) nodes remove the values that are acknowledged by the sink from their aggregation vector (see point 2) so that they are not sent to the sink again and (ii) we disallow combining one set of aggregated values with another (see point 3), as this could cause the values in the union of those sets to be aggregated twice.

Corollary 7.1. *Arctium correctly monitors distributive and algebraic aggregate functions on the network.*

From Claims 1 and 2, since every value will be delivered exactly once, *Arctium* can track distributive aggregates (such as *sum* and *count*) as well as algebraic ones (such as *average* and *variance*). Holistic aggregates [18] (such as *median* or *top-k*) are not covered by the method. *Arctium* can also track any function that fits into the GM framework (since GM aggregates the values from different nodes using the *average*).

Claim 7.3. *With high probability, epochs in Arctium are not longer than the ones in Crystal.*

Algorithm 7.1. Packet structure and packet handlers at node U.

```
   // The application payload
 1 struct {
 2 |  Data: value
 3 |  Vector: idBuffer
 4 } Packet;
 5
   // Local variables
 6 U.idBuffer : a local vector of node IDs who's values have been aggregated
 7 U.valueBuffer : a local vector of values that have been aggregated
 8 U.aggregate : the current sum of all aggregated values
 9 U.nodeId: current node's ID
10
11 Function upon reception of packet P during the A-Phase
   |  // Implements Point 2 (§ 7.3.3)
12 |  foreach id ∈ P.idBuffer do
13 |  |  if id ∈ U.idBuffer then
14 |  |  |  remove id and its value from U.idBuffer and U.valueBuffer
15 |  |  |  U.aggregate = Sum (U.valueBuffer)
16 |  |  end
17 |  end
18 |  With probability p, back-off in the next T-Phase
19
20
21 Function upon reception of packet P during the T-Phase
   |  // Implements Point 3 (§ 7.3.3)
22 |  if P.idBuffer.size ==1 and idBuffer.notFull then
23 |  |  add P.idBuffer in U.idBuffer
24 |  |  add P.value in U.valueBuffer
25 |  |  U.aggregate = Sum of values in U.valueBuffer
26 |  end
27 |
   |  // Implements Point 4 (§ 7.3.3)
28 |  if (U has aggregated values) and (P.idBuffer.size ≥ U.idBuffer.size) then
29 |  |  foreach id ∈ P.idBuffer do
30 |  |  |  if id ∈ U.idBbuffer then
31 |  |  |  |  remove id and its value from U.idBuffer and U.valueBuffer
32 |  |  |  |  U.aggregate = Sum (U.valueBuffer)
33 |  |  |  end
34 |  |  end
35 |  end
36 |
   |  // Implements Point 1 (§ 7.3.3)
37 |  if (U has not aggregated any value) and (U.nodeId ∈ P.idBuffer) then
38 |  |  stop trying to transmit
39 |  end
```

In the absence of packet losses, at the end of every T-A pair, exactly one node's packet will reach the sink. In *Arctium* packets contain the aggregated values from at least one node, hence fewer or equal packets are required (less T-A pairs) for all nodes' values to reach the sink. Also, only the nodes that have values to transmit in this epoch participate in aggregation, i.e. we do not introduce new messages on the nodes that would otherwise not have values to send in this epoch. The only case where *Arctium* might introduce extra T-A pairs is when, during a T-phase, all nodes with values to send decide to back-off simultaneously. If N is the number of nodes with values to send in an epoch and p is the back-off probability, the total number of extra T-A pairs introduced this way in an epoch follows a Poisson binomial distribution with success probabilities $p^N, p^{N-1}, ..., p^2, p$. For example, for $p = 0.5$ and $N = 26$ the expected number of extra T-A pairs is close to 1, and the probability of having more than e.g. 4 extra pairs is less than 1.7% (by using Chernoff's bound [12]). Moreover, even if extra T-A pairs exist in an epoch, the reduction in T-A pairs by the use of aggregation is likely to counter their effect as can be seen experimentally in Section 7.4.3.

Now let's also consider packet losses for the cases discussed above. Claim 1 only holds with high probability as there is an unlikely scenario with loss of values if all of the following four conditions hold: (i) a node A has aggregated at least one value V_B from another node B, (ii) node B has delegated that value to A and stopped trying to send it to the sink, (iii) no other node has aggregated V_B and (iv) node A repeatedly fails to reach the sink. This scenario is extremely unlikely, given that all these four conditions must hold simultaneously and, as shown in [14] and later in Section 7.4.2, Crystal achieves high reliability, i.e. packet losses are very rare. Even then, node A can deliver the value at the next epoch, as a last resort.

Claim 2 can still hold in the presence of losses with a simple remedy. Duplicates might arise when a node that holds value V sends it to the sink but fails to receive the acknowledgment for that value, due to packet losses. In this case, value V might reach the sink more than once. This can be remedied if the sink keeps track of the values it has received. If it detects a duplicate value, it can resend the potentially lost acknowledgement. Claim 3 still holds with high probability in the presence of packet losses.

7.4 Evaluation

We implemented Geometric Monitoring and *Arctium* in Contiki [5], a well-known operating system for IoT applications. We targeted the TelosB platform

that supports protocols such as Crystal that rely on synchronous transmissions. In this section, we assess the performance of our design based on its duty-cycle as well as with other metrics defined below.

7.4.1 Experimental methodology

Experiment setup: We run our experiments in two settings, similar to the setup used in [29]: (i) A *full-system evaluation* on the Flocklab [23] testbed. Flocklab consists of 26 TelosB nodes deployed with a four hop topology. Using Flocklab, it is possible to test our design on a real deployment with realistic interference due to the presence of people and Wi-Fi signals. Moreover, Flocklab is, at the moment, the only publicly available testbed that still includes the TelosB nodes that support protocols such as Crystal. (ii) A *full-system simulation* on Cooja [25], a cycle-accurate simulator where the *whole network stack* is simulated in software at every node. We use Cooja to reproducibly uncover trends and insights, which we then validate with deployment in Flocklab. The default topology here is similar to the testbed. We also use Cooja to test larger topologies than the one available in Flocklab (see Section 7.4.3).

Data set and monitoring functions: We use the commonly-used Intel Lab data set [2]. We use the first day of temperature readings from 26 nodes as sources of data for the nodes in the testbed. In the original dataset, nodes take a new reading every 31 seconds. In our experiments, we simulate the same period, i.e. *we scale the duty-cycle results to correspond to a period of 31 seconds*.

We experiment with two monitoring functions: the *variance* (also used in [28]) and the *average* of the temperature readings and use different values for the threshold we want to monitor (see Section 7.4.2). Unless otherwise noted, we will use the *variance* and a threshold $T = 2°C^2$.

Configuring Crystal: Crystal has many knobs that allow the protocol to operate on different topologies and network conditions, e.g. one-hop vs multi-hop networks, noisy vs interference-free networks. Those knobs also offer a configurable trade-off between the performance requirements, i.e. allow the user to favor energy efficiency in place of reliability and vice versa. We refer to the original publications of Crystal [13,14] where the authors explain the significance and the methodology of choosing correct values for Crystal's parameters. In this work, we simply choose the set of parameter values presented in Table 7.1 that allows Crystal to operate in a reliable manner.

Metrics of interest: A key evaluation criterion is *duty-cycle (DC)*, i.e. the fraction of the total time that the radio is turned on. In some experiments we also report the *lifetime improvement*, i.e. the reduction in duty-cycle achieved through GM. Related to GM, we also report the *communication reduction* of GM

Parameter	Explanation (X is S, T or A)	Value
N_X	number of Glossy transmissions in phase X	4
W_X	the maximum duration of phase X	12 (ms)
R	number of consecutive empty T-A pairs before the epoch is finished	3

Table 7.1: Crystal's parameter values used in experiments.

Method	Comm. Reduction	DC (%)	Lifet. Impr.	Loss Rate (%)
Baseline	1X	1.22	1X	0.34
GM (variance / T=0.5)	3.2X	0.54	2.26	0.03
GM (variance / T=1)	4.1X	0.46	2.65	0.00
GM (variance / T=2)	4.3X	0.44	2.77	0.00
GM (variance / T=3)	5.5X	0.38	3.21	0.10
GM (average / T=25)	87x	0.18	6.78	0.00

Table 7.2: **Full system evaluation** on the Flocklab testbed, using GM on top of Crystal, for different monitoring functions and threshold values.

in terms of the number of updates suppressed by the algorithm; it is a measure of the efficiency of GM, purely from the application point of view. Finally, we also measure the *loss rate*, in terms of the percentage of updates from individual nodes that fail to reach the sink.

Summary of the experiments: The rest of the evaluation section is organized as follows: in Section 7.4.2 we present testbed and simulation experiments that summarize the performance of our Crystal and GM co-design, without using aggregation. In Section 7.4.3 we focus on our aggregation scheme, *Arctium*, and show, through testbed and simulation experiments, how it manages to reduce the average duty-cycle.

7.4.2 Combining GM and Crystal: overall performance

We start with a real-life deployment on the Flocklab testbed, where we monitor the variance and the average of the temperature readings using different threshold values. We compare the performance against a baseline (same as in [27, 29]) that does not run GM and instead sends all measurements to the sink. However, in our case, the baseline uses Crystal as the underlying protocol.

Testbed experiments: Table 7.2 shows the results from the Flocklab de-

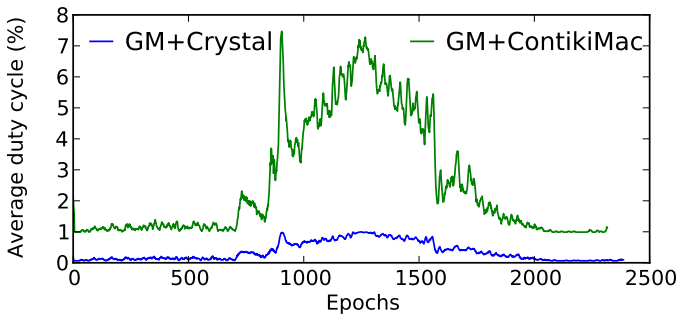


Figure 7.4: Duty-cycle during runtime when monitoring the variance with $T = 2$. The figure also includes results of using GM with ContikiMAC taken from [29].

ployment. The choice of monitoring function and threshold value plays an important role on the effectiveness of the GM and the communication reduction it achieves. When monitoring the variance, GM manages to suppress most of the nodes' values and communicates 3.2 to 5.5 times less than the baseline. What is important in this work though, is that, with the combination of GM and Crystal, that communication reduction is translated as lifetime improvement on the nodes. Our experiments report very low duty-cycle, down to 0.38%. This results in an up to 3.2 times lifetime improvement compared to the baseline, which is using Crystal without GM, an already very energy-efficient protocol. We also note that the system remains fairly reliable, with less than 0.34% loss rate. In the case of the average, GM suppresses most of the communication and the system has a duty-cycle of 0.18%.

Runtime behaviour: We now take a closer look at the dynamic behaviour of the above experiments and the evolution of the average duty-cycle during runtime. We have simulated one of the above experiments in Cooja (we chose the one with $T = 2$) and continuously report the average duty-cycle in Figure 7.4. To highlight the benefits of using GM with Crystal, compared to other network stacks, the figure also includes the equivalent results from [29] where GM is applied on top of ContikiMac [4], a mainstream network stack. The figure shows that, regardless of the protocol stack, GM exhibits a highly dynamic behavior, with periods of low activity where most of the communication is suppressed and periods of high activity (usually when the monitored value is close to the threshold) where GM communicates more. However, the choice of communication protocol has a major effect on the average duty-cycle. Our design that combines GM and Crystal has an order of magnitude smaller duty-cycle and consumes a bare minimum amount of power when communication is

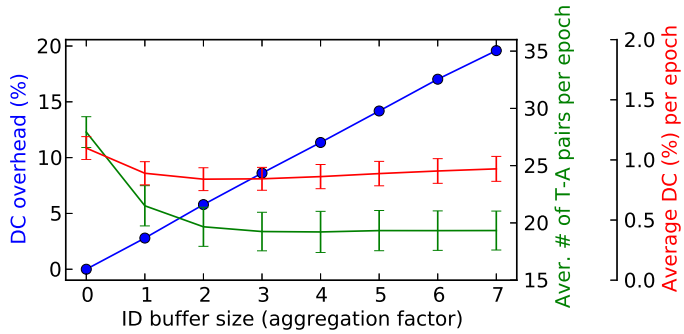


Figure 7.5: Collection of trends illustrating the effect of the aggregation factor (x-axis): (i) the per-packet DC overhead (blue trend), (ii) the average number of T-A pairs per epoch (green trend) and (iii) the average DC per epoch (red trend).

low. On the contrary, mainstream networks stacks have a significant overhead which they have to pay even under low communication [29].

7.4.3 *Arctium*: in-network aggregation under heavy communication

In this section, we focus on our aggregation scheme *Arctium* and show that aggregation is an effective technique to reduce the already low duty-cycle presented in the previous section, even further. In all the experiments of this section, the back-off probability p was set to 0.5.

Effects of the aggregation factor: We start by presenting simulation results that illustrate the effects of the size of the aggregation buffer. As mentioned earlier in Section 7.3, *Arctium* uses a node-ID buffer that is a placeholder for the IDs of the nodes whose values might be aggregated in a single packet. The size of the buffer is a parameter of our design and it affects the effectiveness of the aggregation. In the following experiment, every node in the network tries to send data to the sink (i.e. we adopt the baseline behavior, without the GM algorithm) to illustrate the effect of aggregation. Figure 7.5 shows the evolution of different trends as the size of the ID buffer (aggregation factor) increases in the x-axis. We now explain the significance of each trend.

The blue trend shows the added, per-packet overhead that comes from increasing the buffer size, compared to not having a buffer (aggregation factor 0). Since each packet has to statically reserve space for each entry in the buffer, the size of the packet increases linearly with the size of the buffer. This results in a linear increase in duty-cycle overhead in order to receive and transmit those

	Aggregation factor (size of the buffer)						
	0	1		2		4	
Method	DC	DC	Impr.	DC	Impr.	DC	Impr.
baseline (+ *Arctium*)	1.24%	0.97%	1.27x	0.90%	1.38x	0.93%	1.33x
GM (+ *Arctium*)	0.54%	0.49%	1.10x	0.48%	1.13x	0.50%	1.08x

Table 7.3: **Full system evaluation** on the Flocklab testbed, for different aggregation factor values.

packets. Each added slot in the buffer adds approximately 3% extra overhead.

The green trend illustrates the benefits that come from aggregating values in a single packet: it reports the average number of T-A pairs required to complete an epoch. As the aggregation factor increases, the number of T-A pairs quickly drops, since a bigger buffer allows a packet to carry more information and deliver more values to the sink. Most noticeably, even using a single-slot ID buffer, i.e. going from aggregation factor 0 to 1, is enough to reduce the number of T-A pairs by 23%. The benefit saturates after an aggregation factor of 3 which indicates that, for this given topology, there are not many opportunities to aggregate more than 3 values before the sink receives every node's value.

The red trend reports the average duty-cycle per epoch. This metric factors in both the increasing overhead per packet (blue trend) and the decreasing number of T-A pairs per epoch (green trend). As a result, the average duty-cycle initially decreases and has a minimum when the aggregation factor is 2. At this point, using *Arctium* results in 23% smaller duty-cycle. After that point, the increasing packet overhead dominates and the duty-cycle increases slowly.

Testbed experiments: We support the above simulation results regarding *Arctium* with results from real deployments in the Flocklab testbed. Table 7.3 summarizes the duty-cycle reported for the baseline and GM, as we change the aggregation factor. In the GM experiments, we monitor the variance with a threshold of $T = 0.5$. We also report the improvement in duty-cyle, due to aggregation. For the baseline, the results validate the previous claims: aggregation further reduces the duty-cycle by a varying amount, up to 1.38x. *Arctium* also has a positive effect for the case of GM, where the already low duty-cycle (0.54%) is further decreased by up to 1.13x. Naturally, since GM communicates far less than the baseline, there are fewer messages to send to the sink and fewer chances to aggregate values. Still, the results show that *Arctium* brings improvement even in applications with aperiodic communication patterns.

Testing larger networks: Finally, we experiment with larger and busier

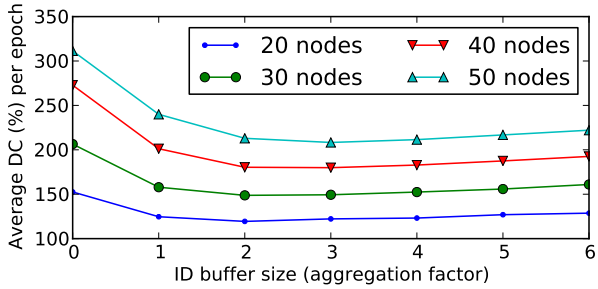


Figure 7.6: The average duty-cycle per epoch for different topologies, as the aggregation factor changes.

topologies in the Cooja simulator. Figure 7.6 reports the average duty-cycle for topologies that range between 20 and 50 nodes. In each topology, every node is trying to send values to the sink. Overall, *Arctium* manages to reduce the duty-cycle at each topology. The reduction ranges from 21% with 20 nodes where there are few packets to aggregate in the network, up to 33% with 50 nodes where there are more opportunities for aggregation.

7.5 Related work

Geometric Monitoring (GM): Since its original publication, GM has been extended in many ways, orthogonal to the design we consider in this work. It has been combined with sketches [10] and prediction models [11] that effectively track aggregates such as join and self-join sizes, while also taking the temporal evolution of the monitored values into account. A summary of the use of GM for query tracking in distributed streaming systems can also be found in [9]. This interest has been motivating also for the work in our paper.

In [16], the authors introduce shape sensitive geometric monitoring, that takes into account properties of the monitored function. Lazerson et al. [20] propose a method to approximate the monitored function to convex/concave components, so that it can be easily checked for violations, providing a good alternative to tackle the complexity of threshold checking we discuss in Section 7.3.2. They also experiment with a high-end embedded platform and show that the method is lightweight. The approximation we propose in our work is orthogonal, in the sense that we leave the function intact and instead bind the local area that nodes have to keep track of.

GM has been studied in the general context of WSN from a high-level perspective. In [28], the authors present an adaption of GM that is designed for

clustered topologies. In [3], GM is used for detecting outliers in the readings of wireless sensor nodes. In both of these lines of work, the network is only considered as a communication abstraction and practical system aspects are not studied. Differently, we take a full system perspective and consider a real network stack. The only work that considers deployment of GM on top of real networks stacks is in [29], which we compare against in Section 7.4.2.

Wireless protocols: Crystal is not the only modern network stack that relies on synchronous transmissions. Ferrari et al. [6] present a protocol that uses synchronous transmissions to support many communication patterns. Landsiedel et al. [19] also use synchronous transmissions and augment them with in-network aggregation to achieve low power and excellent reliability. However, their approach focuses on duplicate-insensitive aggregates (e.g. *min* or *max*) and does not work out of the box for duplicate-sensitive aggregates (e.g. *sum* or *average*). Al Nahas et al. [1] extend the protocol with duplicate-sensitive aggregates. However, all previously mentioned protocols are not designed for cases where traffic is sparse and aperiodic. That is the main motivation why we chose Crystal to build our design upon.

Aggregation: In-network aggregation has been an active topic of research for many years. A large body of work studies aggregation as an application of gossip-based protocols. Friedman et al. [8] discuss different gossiping protocols. Jelasity et al. [15] present a decentralized aggregation protocol based on gossiping. Koldehofe [17] shows the effect of the buffer size in the performance of gossip-based protocols. Kuhn et at. [18] prove bounds and provide algorithms for holistic aggregates, specifically for distributed selection.

In the context of wireless sensor networks, Rajagopala et al. [26] survey different ways aggregation is used in WSNs, mostly by utilizing the network's architecture. Nath et al. [24] show how to approximately track duplicate-sensitive aggregates in WSNs. In this work, we present an approach that builds on top of a modern network protocol that has not been studied before in the context of in-network aggregation.

7.6 Conclusions

We show how a general threshold monitoring framework (GM) can be co-designed together with a state-of-the-art wireless protocol (Crystal), for the problem of continuous threshold monitoring. Detailed results from testbed deployments show that the two approaches complement each other and are able to achieve a very low duty-cycle, up to 10x less than a mainstream network stack, which was the limiting factor in previous work. In particular, we present the

way we orchestrate the communication of GM using the existing schedule of Crystal. We also introduce an efficient approximation for threshold checking that makes this co-design possible. Moreover, we extend our design to also exploit latent opportunities for aggregation in Crystal and improve the lifetime of applications that are monitoring network aggregates. Our aggregation scheme, called *Arctium*, allows nodes to overhear and correctly aggregate neighbouring node's values and manages to further improve the lifetime of the nodes by up to 1.13-1.38x. Our results show that network-stack awareness can bring practical value in applications such as GM for real IoT deployments, especially when coupled with modern network protocols. Our code is available online: `https://github.com/mpastyl/Arctium`.

Acknowledgements

The research leading to these results has been partially supported by the Swedish Civil Contingencies Agency (MSB) through the projects RICS and RIOT, by the Swedish Foundation for Strategic Research (SSF) through the framework project FiC and the project LoWi, by the Swedish Research Council (VR) through the project ChaosNet, and from the European Community's Horizon 2020 Framework Programme under grant agreement 773717.

Bibliography

[1] Beshr Al Nahas, Simon Duquennoy, and Olaf Landsiedel. Network-wide consensus utilizing the capture effect in low-power wireless networks. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, SenSys, 2017.

[2] Peter Bodik, Wei Hong, Carlos Guestrin, Sam Madden, Mark Paskin, and Romain Thibaux. Intel Lab Data, 2004.

[3] Sabbas Burdakis and Antonios Deligiannakis. Detecting Outliers in Sensor Networks Using the Geometric Approach. In *IEEE International Conference on Data Engineering*, 2012.

[4] Adam Dunkels. The ContikiMac radio duty cycling protocol. Technical report, Swedish Institute of Computer Science, 2011.

[5] Adam Dunkels, Björn Gronvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *29th IEEE Int'l Conf. on Local Computer Networks*, 2004.

[6] Federico Ferrari, Marco Zimmerling, Luca Mottola, and Lothar Thiele. Low-power wireless bus. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, SenSys, 2012.

[7] Federico Ferrari, Marco Zimmerling, Lothar Thiele, and Olga Saukh. Efficient network flooding and time synchronization with Glossy. In *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks*, April 2011.

[8] Roy Friedman, Daniela Gavidia, Luis Rodrigues, Aline Carneiro Viana, and Spyros Voulgaris. Gossiping on manets: The beauty and the beast. *SIGOPS Oper. Syst. Rev.*, 41(5), October 2007.

[9] Minos Garofalakis. Approximate geometric query tracking over distributed streams. *IEEE Data Eng. Bull.*, 2015.

[10] Minos Garofalakis, Daniel Keren, and Vasilis Samoladas. Sketch-based Geometric Monitoring of Distributed Stream Queries. *Proc. VLDB Endow.*, 2013.

[11] Nikos Giatrakos, Antonios Deligiannakis, Minos Garofalakis, Izchak Sharfman, and Assaf Schuster. Prediction-based geometric monitoring over distributed data streams. In *ACM SIGMOD International Conference on Management of Data*, 2012.

[12] Torben Hagerup and Christine Rüb. A guided tour of chernoff bounds. *Information Processing Letters*, 33(6):305 – 308, 1990.

[13] Timofei Istomin, Amy L. Murphy, Gian Pietro Picco, and Usman Raza. Data prediction + synchronous transmissions = ultra-low power wireless sensor networks. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems*, SenSys, 2016.

[14] Timofei Istomin, Matteo Trobinger, Amy L. Murphy, and Gian Pietro Picco. Interference-resilient ultra-low power aperiodic data collection. In *Proceedings of the 17th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN, 2018.

[15] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3), August 2005.

[16] Daniel Keren, Izchak Sharfman, Assaf Schuster, and Avishay Livne. Shape Sensitive Geometric Monitoring. *IEEE Trans. Knowledge and Data Eng.*, 2012.

[17] Boris Koldehofe. Buffer management in probabilistic peer-to-peer communication protocols. In *22nd International Symposium on Reliable Distributed Systems, 2003. Proceedings.*, Oct 2003.

[18] Fabian Kuhn, Thomas Locher, and Roger Wattenhofer. Distributed selection: A missing piece of data aggregation. *Commun. ACM*, 51(9), September 2008.

[19] Olaf Landsiedel, Federico Ferrari, and Marco Zimmerling. Chaos: Versatile and efficient all-to-all data sharing and in-network processing at scale. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys, 2013.

[20] Arnon Lazerson, Daniel Keren, and Assaf Schuster. Lightweight monitoring of distributed streams. *ACM Trans. Database Syst.*, 43, July 2018.

[21] Jay Lee, Behrad Bagheri, and Hung-An Kao. A cyber-physical systems architecture for industry 4.0-based manufacturing systems. *Manufacturing Letters*, 3, 2015.

[22] Krijn Leentvaar and Jan Flint. The capture effect in fm receivers. *IEEE Transactions on Communications*, 24(5), 1976.

[23] Roman Lim, Federico Ferrari, Marco Zimmerling, Christoph Walser, Philipp Sommer, and Jan Beutel. Flocklab: A testbed for distributed, synchronized tracing and profiling of wireless embedded systems. In *2013 ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2013.

[24] Suman Nath, Phillip B. Gibbons, Srinivasan Seshan, and Zachary R. Anderson. Synopsis diffusion for robust aggregation in sensor networks. In *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems*, SenSys, 2004.

[25] Fredrik Osterlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, and Thiemo Voigt. Cross-level sensor network simulation with cooja. In *31st IEEE Conf. on Local Computer Networks*, 2006.

[26] Ramesh Rajagopalan and Pramod K. Varshney. Data-aggregation techniques in sensor networks: A survey. *IEEE Communications Surveys Tutorials*, 8, Fourth 2006.

[27] Izchak Sharfman, Assaf Schuster, and Daniel Keren. A geometric approach to monitoring threshold functions over distributed data streams. In *ACM SIGMOD Int'l Conf. on Management of Data*, 2006.

[28] Izchak Sharfman, Assaf Schuster, and Daniel Keren. Aggregate threshold queries in sensor networks. In *IEEE Int'l Parallel & Distr. Process. Symp.*, 2007.

[29] Charalampos Stylianopoulos, Magnus Almgren, Olaf Landsiedel, and Marina Papatriantafilou. Geometric monitoring in action: a systems perspective for the internet of things. In *2018 IEEE 43rd Conference on Local Computer Networks (LCN)*, 2018.

Part V

Appendix

Charalampos Stylianopoulos, Magnus Almgren,
Olaf Landsiedel, Marina Papatriantafilou,
Trevor Neish, Linus Gillander,
Bengt Johansson, Staffan Bonnier

Industry Paper: On the Performance of Commodity Hardware for Low Latency and Low Jitter Packet Processing

An adapted version of the paper that appeared in *the Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems (DEBS)*, ACM 2020.

A

Industry Paper: On the Performance of Commodity Hardware for Low Latency and Low Jitter Packet Processing

Abstract

With the introduction of Virtual Network Functions (VNF), network processing is no longer done solely on special purpose hardware. Instead, deploying network functions on commodity servers increases flexibility and has been proven effective for many network applications. However, new industrial applications and the Internet of Things (IoT) call for event-based systems and midleware that can deliver ultra-low and predictable latency, which present a challenge for the packet processing infrastructure they are deployed on.

In this industry experience paper, we take a hands-on look on the performance of network functions on commodity servers to determine the feasibility of using them in existing and future latency-critical event-based applications. We identify sources of significant latency (delays in packet processing and forwarding) and jitter (variation in latency) and we propose application- and system-level improvements for removing or keeping them within required limits. Our results show that network functions that are highly optimized for throughput perform sub-optimally under the very different requirements set by latency-critical applications, compared to latency-optimized versions that have up to 9.8X lower latency. We also show that hardware-aware, system-level configurations, such as disabling frequency scaling technologies, greatly reduce jitter by up 2.4X and lead to more predictable latency.

A.1 Introduction

High-speed networks are a key part of today's connected world. As the number of connected devices increases [7] and new event-based applications with strict performance requirements (e.g., those related to Industry 4.0 [10] that combines factory automation with communication technologies) become common, next-generation networks face new challenges with respect to scaling and meeting those requirements. In particular, in network architectures, the user-plane[1] [13] i.e. the component that is responsible for carrying and processing network traffic, needs to support reliable connectivity with minimum added delays (latency).

Over the last years, network technologies have shifted from specialized hardware platforms to a user-plane that can be deployed on commodity, off-the-shelf hardware, either natively or using virtualization and container technologies [19].

[1]The term data-plane is also used in the literature, we use the term user-plane throughout the paper.

This shift allows flexibility in the design and deployment of network functions to support a variety of event-driven applications, reduces deployment cost and enables horizontal scaling. It also allows the deployment of Virtual Network Functions (VNF) where packet processing such as switching, firewalls and intrusion detection systems, is decoupled from the hardware that it is deployed on.

While the aforementioned shift to commodity hardware and VNF has been proven successful, e.g., for mobile broadband applications, such network technologies have not yet been put to use in more demanding applications such as Industry 4.0. In particular, one of the requirements of such applications is low latency. In upcoming event-based industrial applications, such as factory automation, machine-to-machine communication and autonomous driving, the network infrastructure is required to deliver low end-to-end latency (less than 10ms in some cases). Another, often overlooked requirement is that variation in latency (jitter) must be kept to a minimum. High jitter might lead to interruptions in service, e.g., in an automated production line where co-operating machinery might experience differences in latency and desynchronize.

The challenge to meet these requirements is particularly pressing for mobile networks that will play a central role in future Industry 4.0 deployments. In particular, the evolution of the core packet processing network, i.e. Evolved Packet Core (EPC) [17], needs to enable future mobile networks to meet the required performance.The packet processing involved in the user-plane of EPC involves many event-based network functions such as Deep Packet Inspection [22] and firewalls, but the bare minimum functionality is packet switching, which we focus on in this paper.

When considering software-based packet processing in general purpose servers, there are several factors that can contribute to high latency. They range from operating system interrupts or scheduling and timesharing with other tasks. Moreover, the packet processing application itself needs to be optimized to make best use of the underlying hardware [23] and focus on delivering low latency. Hence, the performance and feasibility of using commodity hardware for latency-critical event-based applications is still an open issue.

In this industrial experience report, we establish a baseline for the latency and jitter performance of software-based packet processing on commodity hardware. Our goal is to determine the feasibility of using software-based packet processing on commodity hardware for challenging and latency-critical event-based applications in Industry 4.0. As a starting point, we focus on the bare minimum functionality that the user-plane performs, namely packet forwarding. We design and perform experiments to identify sources of unnecessary latency and jitter and we propose ways to remove them, through optimizations in the application

itself and at system level. Specifically, our evaluation shows that:

- In order to come close to the reliable and low latency network requirements of event-based applications, packet processing must be optimized to avoid buffering packets as much as possible, even if that comes at the cost of a reduced sustained throughput.
- System-level, hardware-aware configurations, such as disabling frequency scaling, can have a noticeable effect on latency and jitter.

The rest of the paper is organized as follows. Section A.2 gives the required background on low-latency packet processing. In Section A.3 we identify sources of latency and jitter and show how to mitigate them. We present our experimental methodology in Section A.4 and the results from the experiments in Section A.5. We discuss those results in connection with application requirements in Section A.6. We present related work in Section A.7 and conclude in Section A.8.

A.2 Preliminaries

In this section, we summarize relevant information on the requirements of Industry 4.0 applications, the packet processing involved in the packet core, as well as a brief background on user-space networking.

A.2.1 Ultra Reliable Low Latency Communication Requirements

Ultra Reliable Low Latency Communication (URLLC) is a collection of services supported by the upcoming fifth generation networks (5G), designed to address the needs of latency-critical event-based applications, mainly related to machine-to-machine communication and industrial applications [20].

The basic requirements and characteristics of URLLC services that relate to the network infrastructure they are deployed on are the following: (a) *Low latency*. While there is a plethora of studies mentioning different latency requirements for the same application, typical latency requirements range from 1 msec to 50 msec end-to-end latency. Note that this requirement is about the maximum guaranteed latency and not the average. (b) *High reliability*. Most use cases require a highly reliable network infrastructure that must deliver packets with 99.99% to 99.999% reliability. (c) *Low jitter*. On many industrial applications, jitter (variations in latency) can cause disruptions in service, even if the latency

itself remains within the acceptable bounds. (d) *Low traffic rates*. Fortunately, in most cases, the traffic generated by industrial event-based applications is not both latency- and throughput-critical. Typical use cases generate roughly 50 Mbps or less, which is a very low traffic rate for modern networks.

A.2.2 The Evolved Packet Core

The Evolved Packet Core (EPC) is the core network of Long Term Evolution (LTE) that supports mobile broadband in current mobile telecommunication standards (4G) [17]. It consists of different network functions, such as mobility management, quality of service and lookup of subscription information [4]. The user-plane of EPC is responsible for processing packets between the nodes of the EPC and for connecting them with external networks. The user-plane includes many functions such as firewalls and deep packet inspection, but it is primarily responsible for packet forwarding (switching) [13]. That is the network function that we focus on in this paper.

A.2.3 User-space packet processing

Traditionally, software-based packet processing uses the kernel's network stack to send and receive packets. However, relying on the kernel for packet I/O involves issuing an interrupt every time there is a new packet to be received from the network card, which introduces overheads. Even though newer kernel versions reduce the number of interrupts generated [21], kernel-based networking cannot match the processing speed needed by high performance networks.

In the last few years, user-space solutions for packet processing have become popular, with the *Data Plane Development Kit* (DPDK) being the most common packet I/O framework [2]. In DPDK, packets that are received from the network card are sent directly to memory that is mapped to user-space. DPDK's driver keeps polling the memory for new packets, instead of waiting for an interrupt to be issued. Additionally, DPDK's library includes support for fast packet processing across many cores and heavily relies on batching multiple packets to amortize the per-packet processing cost, making it the de-facto choice for high speed networks.

A.3 Latency and Jitter

In this section, we identify sources of latency and jitter in the context of packet processing on commodity hardware and propose methods to mitigate their effects.

Commodity hardware platforms are generally not designed for low latency and real-time processing. As a result, there are many sources of latency, stemming from, e.g., the scheduler, the operating system or packet processing application itself. While there are many suggestions on how to improve the real-time characteristics of commodity systems [16], we focus on specific parts and measure their effects. Specifically, we target latency and jitter due to: (a) the packet I/O framework and forwarding application and (b) the operating system and the hardware platform itself.

A.3.1 Packet I/O and forwarding application

Choice of Packet I/O: As previously mentioned in Section A.2, kernel-based packet processing introduces overheads. For this reason we use a user-space packet I/O framework (DPDK) and its own simple packet switching application (Layer 2 packet forwarding). We compare its performance with the traditional kernel-based packet I/O (NAPI [21]) using *Linux bridge* [24] for packet switching.

Note that DPDK is designed with high throughput as the primary goal. While the vast majority of that increased throughput comes from reducing the per-packet processing latency, DPDK's design is not necessarily oriented towards maintaining a low per-packet maximum latency. In fact, as we discuss next and experimentally show in Section A.5, the per-packet latency can be high under low load, due to buffering.

Low Traffic Rate + Buffering = High Latency: For the DPDK version, as a staring point, we use the Layer 2 forwarding example application provided by DPDK's library, that simply forwards packets from one port to another. By default, the Layer 2 forwarding application will receive and buffer up to 32 packets before sending them to the outgoing port as a single batch. This reduces the number of times the application communicates with the network card and helps sustain a high processing rate when the incoming packet rate is high. However, buffering can severely impact latency at low traffic rates. Recall from Section A.2 that URLLC applications usually have low traffic rates, so maintaining low latency under such conditions is critical. When the input rate is low, there is a significant delay until the buffer is full of packets and can be sent to the network card which causes: (a) high latency on the first packet that is buffered, since it has to wait for 31 more packets to arrive and be processed and (b) high variation in latency, due to the difference in waiting time between the first packet that gets buffered and the last one before the buffer is flushed. This effect on DPDK's Layer 2 forwarding application has been reported by Kawashima et al. [11].

Since our target applications require very low latency at low traffic rates, we mitigate the effects of buffering and sacrifice the performance at high traffic rates for low and consistent latency at low rates (in Section A.6 we quantify experimentally how much throughput gets sacrificed). We set the size of the buffer to one packet to send every received packet to the outgoing port as soon as it is received. An alternative approach would be to set a timer that will flush the packet buffer at fixed intervals. We choose to disable buffering in order to come as close as possible to the lowest possible latency that we can achieve. We study the effect that disabling buffer has in latency and jitter in Section A.5.

A.3.2 Operating system and hardware platform

We next identify and tackle sources of latency and jitter that come from the operating system and its interaction with the hardware platform. The goal is to ensure (to the extent that it is possible) that the user-space application handling the packet processing does not get any interruptions in its service.

Nadathur et al. [16] suggest multiple kernel options that can contribute to lower and more stable latency, including real-time kernel patches specifically for this purpose. We follow several of these considerations and introduce additional ones. Specifically:

- **Thread isolation:** We isolate the cores that are used by DPDK from the kernel scheduler, to ensure that no other task will timeshare or use resources from those cores.
- **Disable interrupt balancing:** We disable the daemon that dynamically distributes interrupts to cores, to avoid handling unrelated interrupts by the cores running DPDK.
- **Disable turbo-boost:** Turbo-boost is a technology used in Intel CPUs that allows scaling the frequency of a core dynamically during peak loads, even beyond the nominal values, if the power and temperature budget allows [9]. We find (and experimentally show in Section A.5) that on some platforms, enabling turbo boost causes high variation in packet processing latency, possibly due to interruptions in processing involved when the CPU frequency changes.

A.4 Experimental methodology

In this section, we present our experimental methodology. We describe the hardware platform used in our experiments, the network topology and the way

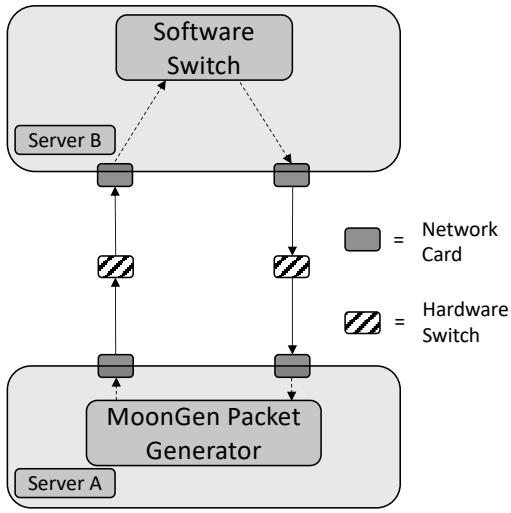


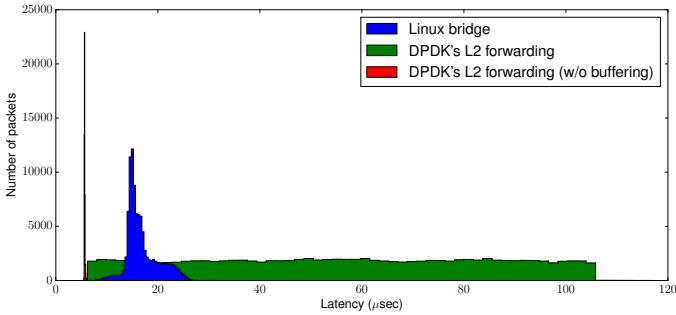
Figure A.1: The experimental setup.

test traffic is generated. We also present the metrics that are relevant in our evaluation.

Hardware Platform: We use servers and network infrastructure from CloudLab [3], a platform that allows bare metal access to a wide range of hardware devices. A summary of our experimental setup is shown in Figure A.1. Each of the two servers is a 20-core NUMA platform with Xeon E5 at 2.26 Ghz that supports 2-way hyperthreading. Each server has two Intel 82599ES 10 Gb Ethernet network cards, connected to each other as shown in Figure A.1, through an internal network of hardware switches. Each server also has 1Gb Ethernet card connected to the external network for control over the servers (not shown in Figure A.1).

While it is currently not possible to connect the servers directly, we performed a loop-back test to establish a baseline on the latency introduced by the hardware switches. Our measurements showed an average latency of 1.1 μsec and 0.1 μsec difference between maximum and minimum latency per switch (these measurements include any overhead added from the packet generation software, which we describe next).

Traffic Characteristics and Generation: We use the MoonGen packet generator [5] to send traffic and measure latency and jitter. MoonGen uses DPDK to send and receive traffic and supports measurements with 100 nsec precision. Unless otherwise noted, we generate a low, constant load of 64 byte



(a) Histogram of latency measurements using different packet processing versions. The spike at 5.7 μsec corresponds to the version that does not use buffering and has too low variation in latency to be clearly visible.

	Linux bridge	DPDK	DPDK (w/o buffering)
Average latency (μsec)	16.8	56.1	5.7
Minimum latency (μsec)	7.8	6.2	5.5
Maximum latency (μsec)	45.0	117.0	17.8
Deviation (μsec)	3.1	28.5	0.6

(b) Latency statistics for different packet processing versions.

Figure A.2: Histogram and latency statistics for packet processing of different versions. Different versions have significant differences in both the average and the general distribution of latency.

UDP packets at 100 Kpackets/sec and we measure latency every 1ms. We generate traffic in platform A, send it to platform B where the software switch application forwards it to a different port that is connected back to platform A. Latency is measured using hardware timestamps generated at the network cards.

Metrics: We use latency, reported as the round-trip-time of packets from the moment they leave the network card at server A until they return. We report jitter as the absolute difference in latency between two successive measurement samples [18]. Finally, we also report throughput as the rate at which we send

traffic to the network at server A.

A.5 Empirical study

In this section, we present the results from our experiments and show the effect of our mitigation techniques on latency and jitter.

A.5.1 Packet I/O framework

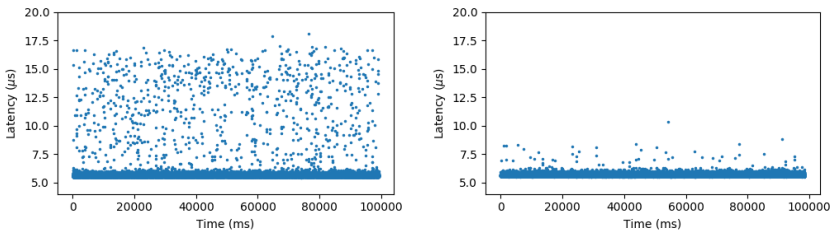
We start by studying the effect that the choice of the packet I/O framework has on latency. In Figure A.2a we show the latency when using kernel-based packet processing (linux-bridge) versus user-space packet processing (DPDK). We include the latency statistics in the table of Figure A.2 for completeness. The latency samples for the linux bridge are spread around an average of 16.8 μsec, with a minimum and maximum latency of 7.8 and 45 μsec respectively. On the other hand, the latency measurements of DPDK's original layer 2 forwarding application are evenly spread between 6.2 and 117 μsec, with an average of 65 μsec. This even spread is clearly an effect of buffering, where the latency of a packet depends greatly on how early or late it was placed on the buffer. As a result, we see that the original DPDK version of packet forwarding is not designed to perform well with respect to latency at low traffic rates.

In summary: Using a high-performance, user-space I/O framework does not, on its own, guarantee low latency at low traffic rates. Next, we show the effect of buffer-removal on latency.

A.5.2 Application layer optimizations

We now present the latency measurements of a modified version of DPDK's L2 forwarding application where packets are sent directly to the outgoing interface after they are received. We include those results in Figure A.2a to compare against the other versions, especially against the version that uses buffering. By disabling packet buffering, the latency of most packets is concentrated at around 5.7 μsec, which is 9.8X lower than than the original version that buffers packets. However, the maximum reported latency was 17.8 μsec, which means that there are still sources of spurious latency spikes, that originate from the underlying hardware and operating system. We discuss their effect and mitigation next.

In summary: Disabling packet buffering in packet forwarding applications is necessary to achieve low latency when the traffic rate is low.



(a) Latency samples before system level configurations.

(b) Latency samples after system level configurations.

	Before change	After changes
Average jitter (μsec)	0.17	0.07
Maximum jitter (μsec)	12.52	4.80
Deviation (μsec)	0.88	0.10

(c) Jitter statistics before and after system level configurations.

Figure A.3: The effects of system level optimizations on latency. The system level configurations lead to more predictable latency that is concentrated around the average values and has less deviation than before applying the changes.

A.5.3 System layer configurations

After optimizing the application for low latency, we now focus on the system level configurations and their effect on latency. Figure A.3a shows a different representation of the latency of DPDK's L2 forwarding version that does not use any buffering, where we plot all the latency samples gathered during our experiments. We see that the vast majority of samples have latency around the average of 5.7 μsec, but there are spurious increases in latency that range up to 18 μsec, indicating that latency increases unpredictably during the execution of our experiments. We also report jitter in the table of Figure A.3. The average jitter is 0.17 μsec with a maximum reported jitter of 12.52 μsec.

In Figure A.3b, we show the net effect of applying the system configurations presented in Section A.3. The effect of those configurations is immediately noticeable in this representation, where the vast majority of spurious increases in latency have been mitigated. The average reported jitter is 0.07 μsec, which is 2.4X lower than before applying the system level configurations. Performing longer runs, we find that the maximum latency is not affected (our maximum reported latency was 20 μsec), but latency is much more predictable and concentrated around the average.

In summary: System layer configurations such as thread isolation and disabling turbo boost significantly reduce jitter.

A.5.4 Latency vs throughput

The mitigation techniques we introduced in Section A.3 and evaluated so far focus on minimizing latency at low traffic rates. However, they come at a cost: the maximum sustained packet processing rate is reduced.

In Figure A.4 we report the average and maximum latency of both the original (throughput-optimized) and the latency-optimized packet forwarding DPDK application as we increase the rate at which we generate traffic. When the traffic rate is low, the original version that focuses on throughput has high average and maximum latency, since it takes long for the packet buffer to fill. As the traffic rate increases, the effect of buffering on latency decreases and this version maintains a low latency at high rates. In fact, the original DPDK L2 forwarding application can easily support the maximum available bandwidth at the link (10 Gbps). The version that does not do any buffering has low latency at low traffic rates, which gradually increases with the traffic rates, until roughly 4.3 Gbps. After that point, this version cannot process packets at the same rate as they arrive. As a result the packet queues start to fill, latency increases and we start to see packets being dropped. However, as we discuss in Section A.2.1, it is the low-throughput end (often less than 50Mbps) that is relevant for URLLC applications. At those low rates, our latency mitigation techniques manage to keep the latency an order of magnitude lower than what it originally was.

A.6 Discussion

The requirements of the URLLC on latency and jitter presented in Section A.2, relate to the end-to-end application requirements and include the overhead of many components of the network, including wireless communication with the base station. As such, for an application with e.g. a 10ms latency requirement budget, only a small part of that budget can be allocated to the user-plane of the packet core and specifically to packet switching. However, it is hard to judge exactly how much the target latency for packet switching alone would be. In our experiments, the maximum reported latency was 20 μsec, which is likely to be sufficiently low for most industrial applications. However, since we examine only a small part of the network stack involved, it is hard to form a clear picture of the feasibility of deploying a network stack on commodity hardware for latency critical applications. Moreover, as we demonstrated in Section A.5.4, our

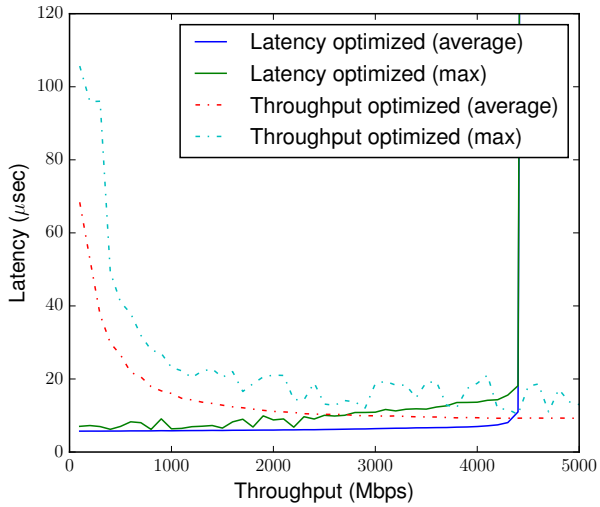


Figure A.4: Average and maximum latency at different packet transmission rates. The latency-optimized version cannot sustain the incoming packets rates after roughly 4.3 Gbps.

latency reduction techniques come at the cost of lower sustainable throughput at high rates, but are a good fit for URLLC applications that have low traffic rates.

A.7 Related work

In this section, we present related work on the topic of evaluating the packet processing performance of commodity hardware.

Emmerich et al. [6] perform extensive benchmarks using different packet I/O frameworks and identify performance bottlenecks in hardware and software. Gallenmüller et al. [8] focus on the performance of I/O frameworks and present an analytical model to predict their performance. They also experiment with packet buffering and show its effect on latency, but only at high traffic rates. Kawashima et al. [11] evaluate the performance of packet forwarding applications across many packet I/O frameworks and execution environments, focusing on both throughput and latency. They also identify that the original DPDK L2 forwarding application uses buffering which negatively affects latency at low traffic rates.

A large body of research evaluates commodity hardware in the context of NFV. Anderson et al. [1] evaluate packet I/O frameworks in different environ-

ments, including virtual machines and containers. They provide results on latency and jitter, but do not use fast, user-space I/O frameworks such as DPDK. Kourtis et al. [12] evaluate the processing throughput of deep packet inspection applications with DPDK on both bare-metal and virtualized environments.

Focusing on mobile broadband networks, Lange et al. [14] evaluate the performance a Serving Gateway that involves both the user-plane and control plane events, and show that user-space networking based on DPDK can greatly improve the per-packet processing time. Mao et al. [15] study the performance of a software-based Radio Access Network under strict latency requirements and show that light-weight virtualization with containers, together with frameworks like DPDK can lead to worst-case latency that is within the required bounds of latency-critical applications. Contrary to those approaches, in this paper we show the trade-off between latency and throughput, study the sources of latency and jitter and show system and applications configurations that can reduce them.

A.8 Conclusions

In this paper, we consider the performance of packet processing deployed on commodity hardware with respect to latency and jitter and propose a baseline on the feasibility of such platforms for the packet processing needs of Industry 4.0 applications. We identify sources of latency and jitter in the packet processing application as well as the underlying system and show ways to mitigate them. Specifically, we show that optimizing applications for latency rather than throughput (e.g. by disabling buffering of packets), greatly reduces average latency by up to 9.8X, at low traffic rates, which is important for event-based URLLC that usually have low volumes of traffic but are sensitive to latency. Moreover, we show that system level configurations, such as disabling dynamic frequency scaling makes latency more predictable and reduces jitter.

Acknowledgements

The research leading to these results has been partially supported by the Swedish Civil Contingencies Agency (MSB) through the projects RICS and RIOT, by the Swedish Foundation for Strategic Research (SSF) through the framework project FiC, by the Swedish Research Council (VR) through the project ChaosNet and the project AgreeOnIT, the Vinnova-funded project “KIDSAM”, and from the European Community’s Horizon 2020 Framework Programme under grant agreement 773717.

Bibliography

[1] Jason Anderson, Hongxin Hu, Udit Agarwal, Craig Lowery, Hongda Li, and Amy Apon. Performance considerations of network functions virtualization using containers. In *2016 International Conference on Computing, Networking and Communications (ICNC)*, pages 1–7, Feb 2016.

[2] DPDK. Data plane development kit. `https://www.dpdk.org`, 2019.

[3] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[4] Romaric Duvignau, Marina Papatriantafilou, Konstantinos Peratinos, Eric Nordström, and Patrik Nyman. Continuous distributed monitoring in the evolved packet core. In *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*, pages 187–192, 2019.

[5] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *Internet Measurement Conference 2015 (IMC'15)*, Tokyo, Japan, October 2015.

[6] Paul Emmerich, Daniel Raumer, Florian Wohlfart, and Georg Carle. Assessing soft-and hardware bottlenecks in pc-based packet forwarding systems. *ICN 2015*, page 90, 2015.

[7] Ericsson. Internet of things forecast. `https://www.ericsson.com/en/mobility-report/internet-of-things-forecast`, 2016. Accessed: 2019-01-15.

[8] Sebastian Gallenmüller, Paul Emmerich, Florian Wohlfart, Daniel Raumer, and Georg Carle. Comparison of frameworks for high-performance packet io. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 29–38. IEEE, 2015.

[9] Intel. Higher performance when you need it most. `https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html`, 2019.

[10] Nasser Jazdi. Cyber physical systems in the context of Industry 4.0. In *2014 IEEE International Conference on Automation, Quality and Testing, Robotics*, pages 1–4, May 2014.

[11] Ryota Kawashima, Hiroki Nakayama, Tsunemasa Hayashi, and Hiroshi Matsuo. Evaluation of forwarding efficiency in nfv-nodes toward predictable service chain performance. *IEEE Transactions on Network and Service Management*, 14(4):920–933, Dec 2017.

[12] Michail Kourtis, George Xilouris, Vincenzo Riccobene, Michael Mcgrath, Giuseppe Petralia, Harilaos Koumaras, Georgios Gardikis, and Fidel Liberal. Enhancing vnf performance by exploiting sr-iov and dpdk packet processing acceleration. 11 2015.

[13] James Kurose and Keith Ross. *Computer networks and the internet. Computer networking: A Top-down approach*. London: Pearson, 2016.

[14] Stanislav Lange, Anh Nguyen-Ngoc, Steffen Gebert, Thomas Zinner, Michael Jarschel, Andreas Köpsel, Marc Sune, Daniel Raumer, Sebastian Gallenmüller, Georg Carle, and Phuoc Tran-Gia. Performance benchmarking of a software-based lte sgw. In *2015 11th International Conference on Network and Service Management (CNSM)*, pages 378–383, Nov 2015.

[15] Chen-Nien Mao, Mu-Han Huang, Satyajit Padhy, Shu-Ting Wang, Wu-Chun Chung, Yeh-Ching Chung, and Cheng-Hsin Hsu. Minimizing latency of real-time container cloud for software radio access networks. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 611–616, Nov 2015.

[16] Sundar Nadathur and Jiming Sun. Nfv-i host configuration for low latency. `https://01.org/packet-processing/blogs/nsundar/2018/nfv-i-host-configuration-low-latency`, 2018.

[17] Magnus Olsson, Catherine Mulligan, Shabnam Sultana, Stefan Rommer, and Lars Frid. *EPC and 4G packet networks: driving the mobile broadband revolution*. Academic Press, 2013.

[18] S Poretsky, J Perser, S Erramilli, and S Khurana. RFC 4689–terminology for benchmarking network-layer traffic control mechanisms. *IETF, October*, 2006.

[19] Ericsson Technology Review. Cloud-native application design in the telecom domain. `https://www.ericsson.com/en/ericsson-technology-review/archive/2019/cloud-native-application-design-in-the-telecom-domain`, 2019.

[20] Joachim Sachs, Gustav Wikstrom, Torsten Dudda, Robert Baldemair, and Kittipong Kittichokechai. 5g radio network design for ultra-reliable low-latency communication. *IEEE network*, 32(2):24–31, 2018.

[21] Jamal Hadi Salim. When NAPI comes to town. In *Linux 2005 Conf*, 2005.

[22] Charalampos Stylianopoulos, Magnus Almgren, Olaf Landsiedel, and Marina Papatriantafilou. Multiple pattern matching for network security applications: Acceleration through vectorization. In *2017 46th International Conference on Parallel ProceWsing (ICPP)*, pages 472–482, Aug 2017.

[23] Charalampos Stylianopoulos, Simon Kindström, Magnus Almgren, Olaf Landsiedel, and Marina Papatriantafilou. Co-evaluation of pattern matching algorithms on iot devices with embedded gpus. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, page 17–27, New York, NY, USA, 2019. Association for Computing Machinery.

[24] Nuutti Varis. Anatomy of a linux bridge. In *Proceedings of Seminar on Network Protocols in Operating Systems*, page 58, 2012.